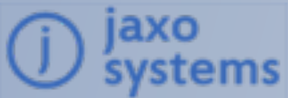
This file contains 4 different resources about running Rexx on Android.

They occur back-to-back in this file --

1. Pierre G. Richard’s original presentation from RexxLA Symposium 2011 on Rexxoid
2. “Rexxoid: Running Rexx on Android Systems” by Julian Reindorf, 2014
3. “REXXOID: Running Rexx on Android” by Julian Reindorf, 2015, RexxLA Symposium
4. “BRexx: Running Rexx on Android Systems” by Eva Gerger, 2015

Rexx for Android

Pierre G. Richard



2011 International Rexx Symposium

December 4-7, 2011 - Aruba, Dutch West-Indies.

Foreword

- Embedded systems are (were?) a programming environment challenge, and, as such, raised our interest while willing to proove the continued usefulness of the REXX concept
- Initially targetted for PalmOS, written in C++, Jaxo REXX library runs on the most widely used operating systems
- The port to Android took 11 days for the core part and the base GUI

jaxo systems

Why Android?

- The success of mobile devices generated a plague of operating systems, inheriting the past, and poorly ”adapted” to the new features.
- For programmers, Android is a *revolution*, benefitting of years of technology evolution in terms of:
  - portability (Java and C, Linux kernel)
  - descriptive GUI (XML) and data storage (SQL),
  - multi-tasking, connectivity and media support
  - coordination of software resources (activities, providers, intents, …)
- Android is the ideal candidate for a glue language as powerful as REXX

jaxo systems

How it works

jaxo systems

A Few Specifics(*)

- It already talks:

  Call Charout "Speaker:", "Bonjour Aruba."

- It properly passes commands to the system:

  "ls –a" /* a typical Linux command (dir) */
  /* running an intent */
  "am start –a android.intent.action.MAIN –n com.android.settings/.Settings"

- It is enough fast:

```
----- REXXCPS 2.2 -- Measuring REXX clauses/second -----
REXX version is: REXX-Jaxo-1.275 4.03 5 Aug 2011
    System is: ANDROID
    Averaging: 10 measures of 30 iterations

   Performance: 431563 REXX clauses per second
```

(device is a Samsung Galaxy Space II running Android 2.3.3)

(*) probably more, to come later – it's an 'own-time' project!

Download and Install

- Nowadays, distributed as an APK (application package file)
  - Run on Android phones and tablets, emulator...
  - Send me mail... (pgr@jaxo.com)

- Soon to be on the Android Market

jaxo systems

Q & A

jaxo systems

Vienna University of Economics and Business

Institute for Management Information Systems

Course 0208, Projektseminar aus Wirtschaftsinformatik (Schiseminar)

Rexxoid: Running Rexx on Android Systems

Julian Reindorf, 1151548

Supervisor: ao. Univ.Prof. Mag. Dr. Rony G. Flatscher

December 25, 2014

Declaration of Authorship

"I do solemnly declare that I have written the presented research thesis by myself without undue help from a second person others and without using such tools other than that specified.

Where I have used thoughts from external sources, directly or indirectly, published or unpublished, this is always clearly attributed.

Furthermore, I certify that this research thesis or any part of it has not been previously submitted for a degree or any other qualification at the Vienna University of Economics and Business or any other institution in Austria or abroad."

Date:

Signature:

Contents

1 Introduction **1**

1.1 Android 1

1.2 Rexx 2

2 Rexxoid **3**

2.1 Getting Started 3

2.2 Permissions 4

2.2.1 Adding New Permissions 4

2.2.2 Permission: INTERACT_ACROSS_USERS_FULL 5

2.3 Debugging 5

2.4 Shell Commands 6

2.5 Shell Extra Values 7

3 Nutshell Examples **8**

3.1 Hello World 8

3.2 Setting Alerts 8

3.3 Adding Calendar Events 11

3.4 Command Line 14

3.5 Creating Emails 15

3.6 Opening Google Maps 17

3.7 Using the Speaker 19

3.8 Reading and Writing Files 19

3.9 Sending Keyevents 20

3.10 Opening the Browser 21

4 Limitations of Rexxoid **23**

5 Rexxoid versus BRexx **24**

5.1 Comparison 24

5.2 Recommendation 26

6 Conclusion **27**

List of Figures

1 The Rexxoid menu . . . 3
2 Rexxoid screens . . . 4
3 Hello World example . . . 8
4 Alert creation - user interaction . . . 10
5 Alert created . . . 10
6 Calculating milliseconds without specified precision . . . 12
7 The date(T) error . . . 13
8 Event creation - user interaction . . . 13
9 Event creation - settings . . . 14
10 Command line showing uptime . . . 15
11 Rexxoid created email . . . 16
12 User chooses location . . . 18
13 Google Maps showing Sydney . . . 18
14 Text read from file . . . 20
15 Keyevent volume . . . 21
16 User chooses url . . . 22
17 www.cnn.com . . . 22

List of Tables

1 Datatypes for intents . . . 7
2 Comparison of Rexxoid and BRexx . . . 26

Listings

1 Hello World . . . 8
2 Create an alert . . . 9
3 Create calendar event . . . 11
4 Send commands to Android shell . . . 14
5 Create email . . . 15
6 Open Google Maps . . . 17
7 Use speaker . . . 19
8 Write to and read from file . . . 19
9 Send keyevent . . . 20
10 Open the browser . . . 21
11 Open Google Maps and zoom in . . . 23

Abstract

Rexx runs on several operating systems, including Windows, MacOS and Linux. However, the recent development towards mobile devices leads to the need of running Rexx also on the most wide-spread mobile device operating system: Android. The Rexxoid interpreter enables Android devices to execute Rexx scripts. Furthermore commands can be sent to the Android shell. This paper describes Rexxoid and presents several short examples. In addition, differences to another approach of running Rexx on Android (BRexx) are discussed.

1 Introduction

This section briefly introduces Android, Rexx and Rexxoid. The afterwards following sections will discuss:

- The Rexxoid application (section 2)
- Rexxoid nutshell examples (section 3)
- Limitations of Rexxoid (section 4)
- Comparison of Rexxoid and BRexx (section 5)

1.1 Android

Android, an operating system (OS) for mobile devices, was first announced in 2007. It is developed by the Open Handset Alliance, which is a multinational team of IT companies, founded by Google Inc [Open14]. The success story of this OS is amazing. Android showed an incredible growth within the quarters between Q1 2009 and Q3 2010. In this timeframe, sales grew from 0 (Q1 2009) to more than 8 million (Q3 2010). By this it outran sales of Blackberry and Apples iOS, thereby becoming the first real threat for the still growing iPhones sales. [see Butl11, 4]

Recent statistics show that Android reached a market share of about 85% in Q3 2014. iOS lags far behind with its approximately 12%. Other competitors struggle hard to stay in the market, sharing the last few percent. [see Inte14]

Android is running on a modified Linux kernel and comes with many built-in activities. Its store is free, in contrast to Apples app store. As a result, applications can be published easily. Using Android, users can control the security settings on their own. Applications need permission to use services of the device. [see Butl11, 5]

Smartphones have drastically changed the way people use their phones. While, in earlier times, phones were used for calls and sending sms only, now a smartphone is an allround device, assisting people in all areas of their lives. Android heavily contributed to this change. [see Butl11, 5]

1.2 Rexx

The programming language Restructured Extended Executor (Rexx) was initiated in 1979 by Mike F. Cowlishaw to provide an easier language for IBM mainframes than Exec 2. The easy to understand language was developed further and in 2005 Open Object Rexx (ooRexx) was published by the Rexx Language Association. [see Flat13, iii f.]

The main features of ooRexx are: [see Flat13, iv ff.]

- It is "backward compatible". Therefore, normal Rexx scripts run unchanged under ooRexx.
- As its name says, it is object oriented. Many useful classes are included out of the box.
- Multithreading is possible.
- It is a fast interpreter.
- ooRexx comes with a comprehensive and detailed documentation with a lot of examples.
- It is free and open source.
- It runs unchanged on 32 and 64 bit operating systems.

The Bean Scripting Framework for ooRexx (BSF4ooRexx) package provides the possibility to camouflage Java objects as Rexx objects. This package increases the possible number of applications for ooRexx even further. [see Flat13, 159 f.]

2 Rexxoid

Rexxoid, a Rexx interpreter, was published by Pierre Richard [see Goog14]. It comes with an Android application which provides a simple text editor and the possibility to execute Rexx code. Additionally, commands can be sent to the Android shell. This enables the user to start activities and therefore build scripts to automate processes on Android devices. Unfortunately, there is no documentation for Rexxoid and the number of users working with it seems to be very low.

2.1 Getting Started

The Rexxoid apk file can be downloaded to one's Android device from the Google Play Store, it is named "Rexx for Android", developed by Jaxo, Inc. After installation, there are a few example scripts that can be executed (see Figure 1). Most of them include only Rexx code and do not send commands to Android itself. One example is more interesting: the "Android System Test" (see Figure 2a and Figure 2b for the editor and execution view of the script). It uses the application manager and opens the settings GUI. If errors arise when executing the script, they will be displayed in the output field.

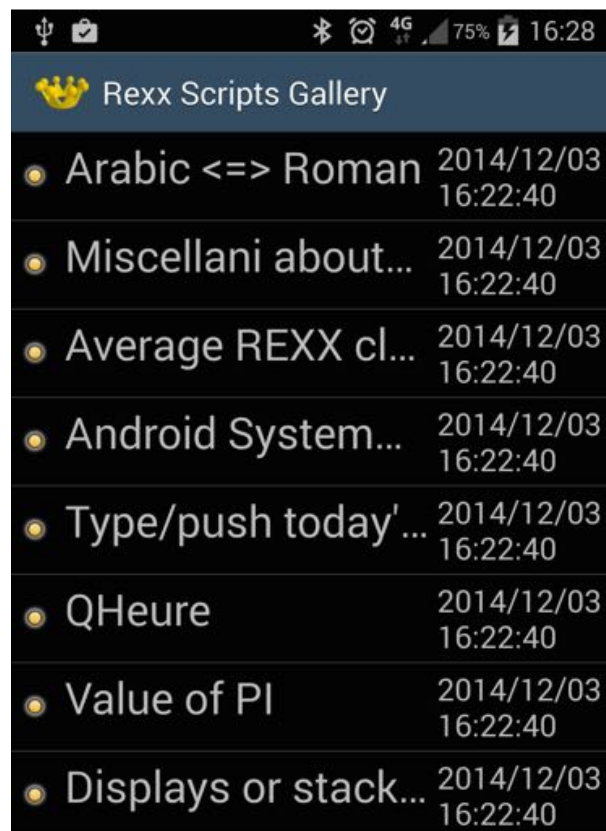
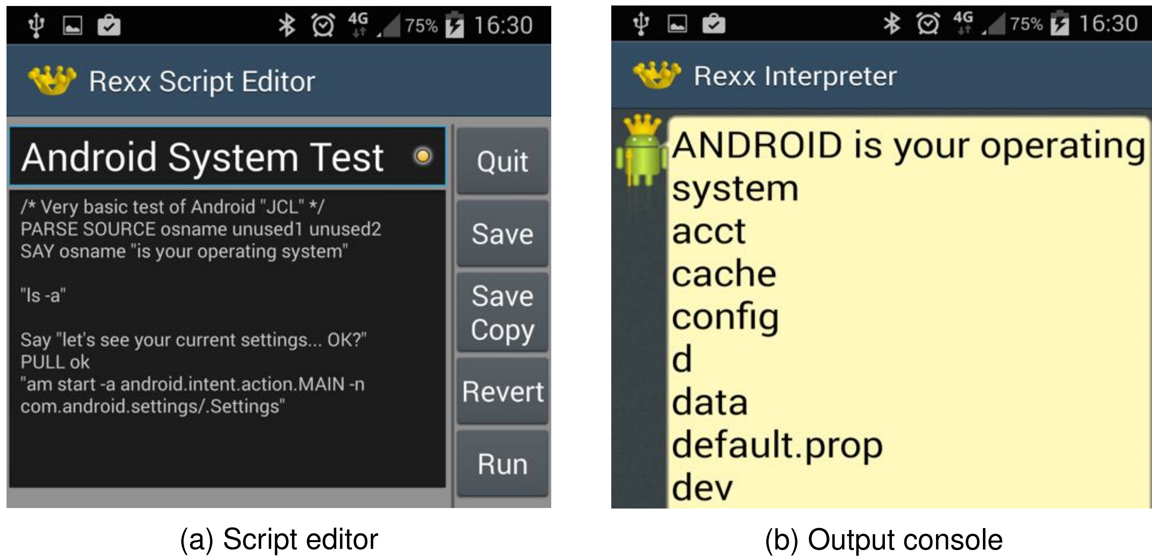


Figure 1: The Rexxoid menu



(a) Script editor (b) Output console

Figure 2: Rexxoid screens

2.2 Permissions

For the execution of some commands, the Rexxoid application needs additional permissions. Section 2.2.1 describes how to grant additional permissions, while section 2.2.2 discusses a frequently thrown permission exception.

2.2.1 Adding New Permissions

Like every other Android application, Rexxoid has a Manifest file where every used permission is stated explicitly. Therefore, every script that runs in Rexxoid has the same permissions as the application itself. By default, only two permissions are used:

- android.permission.READ_EXTERNAL_STORAGE
- android.permission.WRITE_EXTERNAL_STORAGE

That means, if a script calls an intent (or sends any other command to the Android shell) where additional permissions are needed, this will rise a permission denied exception. In the Android operating system an intent is a messaging object, which allows to trigger specific functions of applications [see Wiki14]. Therefore, intents can e.g. be used in Rexxoid scripts to launch other applications.

To modify the permissions the Rexxoid project needs to be downloaded to a computer, its Manifest file needs to be modified and then it must be reinstalled on the mobile device.

To do so, the Rexxoid project needs to be downloaded (e.g. from GitHub: `https://github.com/Jaxo/yaxx/tree/master/android`) first. Next, the downloaded project has to be opened in a suitable IDE (Integrated development environment) for Android programming, e.g. Eclipse with the ADT (Android Development Tools) plugin installed. After that, the AndroidManifest.xml file has to be edited. For each desired permission one line needs to be added (e.g. `<uses-permission android:name="com.android.alarm.permission.SET_ALARM" />` for the permission to create alarms). After all desired permissions are set, the device (i.e. smartphone or tablet) must be connected with a USB cable to the computer. USB debugging (an option in the device's developer options) has to be enabled on the target device. Now, the modified Rexxoid application can be reinstalled via the USB cable from the computer.

If there is no effect, one should try to uninstall the Rexxoid application first, and then install it via Eclipse again. Either way, all scripts should be kept in a backup, because reinstalling the application will delete all of them. In the Rexxoid project, there is a folder "assets", which includes the folder "rexx". Any script in this folder will be available on the device after installation. Note that the name of a script stored in this directory is irrelevant, however, the first line must be a comment, which will then (on the device) be displayed as the scripts name.

2.2.2 Permission: INTERACT_ACROSS_USERS_FULL

Many commands lead to the following error:

```
java.lang.SecurityException: Permission Denial: startActivity asks to run as
 user -2 but is calling from user 0; this requires
android.permission.INTERACT_ACROSS_USERS_FULL
```

Although the error message says so, adding the `android.permission.INTERACT_ACROSS_USERS_FULL` permission to the Android Manifest does not help in this case. Instead, adding the optional parameter `--user 0` to the executed command fixes the problem.

2.3 Debugging

If errors occur, they will be displayed in the output area. Since these are mostly very long messages, it can be difficult to read them. However, the log can be viewed from a computer as well. For doing so, the device and the computer have to be connected with a USB cable. Furthermore, USB debugging needs to be enabled. Then, if not already done, the path to the adb (Android debug bridge), which comes with the installation of the Android Software Development Kit [see Andr14b], has to be added to the $PATH

variable. After that, the terminal can be used to read the logcat by typing `adb logcat` in it. This command will lead to a lot of messages (not only logged by Rexxoid). To filter messages, type e.g. `adb logcat | grep -i "rexx"` (this will display only messages that include "rexx", case insensitive). These steps apply to a Mac OS environment and may differ on other operating systems.

2.4 Shell Commands

Any code enclosed by quotation marks (e.g. "code") that is not an argument to a Rexx function, will be sent to the Android shell.

Commands using the application manager ("am") follow this pattern:

```
am <command> <mandatory arguments> <optional extra values>
```

- "am" stands for application manager.
- "<command>" is to be replaced by the desired command.
- "<mandatory arguments>" is to be replaced by e.g. the intent to be started.
- "<optional extra values>" can be left empty or be replaced by specific extra values.

One important command is the "start" command. It is used to start an activity and follows this pattern:

```
am start -a <full qualified intent name> <optional extra values>
```

- "start" is the command to be executed.
- "-a <full qualified intent name>" specifies the intent action.

For a list of all available commands, please see
`http://developer.android.com/tools/help/adb.html` [see Andr14a].

2.5 Shell Extra Values

Most commands accept additional parameters which have to be defined by their datatype. These parameters follow the pattern:

```
<option> <key> <value>
```

See Table 1 for some of the most important datatypes.

Description	Option	Example
String	--es	--es <key> 'hello World'
Boolean	--ez	--ez <key> true
Integer	--ei	--ei <key> 5

Table 1: Datatypes for intents

For a list of all available extra values that can be added, see `http://developer.android.com/tools/help/adb.html` [see Andr14a].

3 Nutshell Examples

In this section nutshell examples, using the concepts described above, will be shown and described.

3.1 Hello World

The first example traditionally is a program that simply prints the string "Hello World". To follow this intention, here is Hello World in Rexxoid:

```
1 SAY "Hello World"
```

Listing 1: Hello World

As can be seen, this short program runs unchanged on a desktop computer and on an Android device running Rexxoid. Figure 3 shows the output of the program.

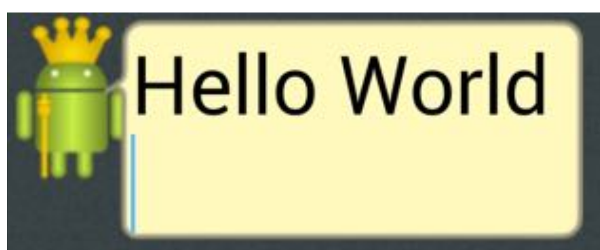


Figure 3: Hello World example

3.2 Setting Alerts

The following example asks the user for some settings and subsequently creates a corresponding alert.

This example requires the following permission (see section 2.2 for more information): `com.android.alarm.permission.SET_ALARM`.

```
SAY "*Alert Creation*"
SAY "Hour?"
PULL hour
SAY "Minute?"
PULL minute
SAY "Message?"
PULL message
"am start -a android.intent.action.SET_ALARM --user 0 "      ,
    "--ei android.intent.extra.alarm.HOUR " hour              ,
    "--ei android.intent.extra.alarm.MINUTES " minute         ,
    " -e  android.intent.extra.alarm.MESSAGE '" message "'"   ,
    "--ez android.intent.extra.alarm.VIBRATE false "          ,
    "--ez android.intent.extra.alarm.SKIP_UI true"
SAY "Done!"
```

Listing 2: Create an alert

Line 1 informs the user about the purpose of the script. Lines 2 to 7 are simple Rexx code that ask the user for some settings. The next command (lines 8 to 13) is sent to the Android shell. The last one informs the user that the script has finished.

Line 8 tells the activity manager (am) to start an activity. There are some additional options which are described below.

- **-a android.intent.action.SET_ALARM**: Specifies the action. This needs to be the fully qualified intent name.
- **--user 0**: Specifies the user (Without this, there will be an exception).
- **--ei android.intent.extra.alarm.HOUR hour**: Sets the alarm hour to the value of `hour`. Must be an integer value.
- **--ei android.intent.extra.alarm.MINUTES minute**: Sets the alarm minutes to the value of `minute`. Must be an integer value.
- **-e android.intent.extra.alarm.MESSAGE message** : Sets the alarm message to the value of `message`. Must be a quoted string value.
- **--ez android.intent.extra.alarm.VIBRATE false**: Specifies that the alarm plays a sound and that the device does not vibrate. If set to true, there will be no audio notification, instead the device only vibrates.

- **--ez android.intent.extra.alarm.SKIP_UI true**: Defines that the alarm user interface should not be shown. If omitted or set to false, the alarm UI will be opened after alarm creation.

Figure 4 shows the script in action. The user wants to create an alert at 23:20 with the message "Go to bed!". As can be seen, Rexxoid prints system messages on activity start. Whenever an activity is started, a TRACE-statement, which contains the return code of Android, is printed (here, "+++ RC(2043491636) +++"). The script ends with the "Done!" statement.

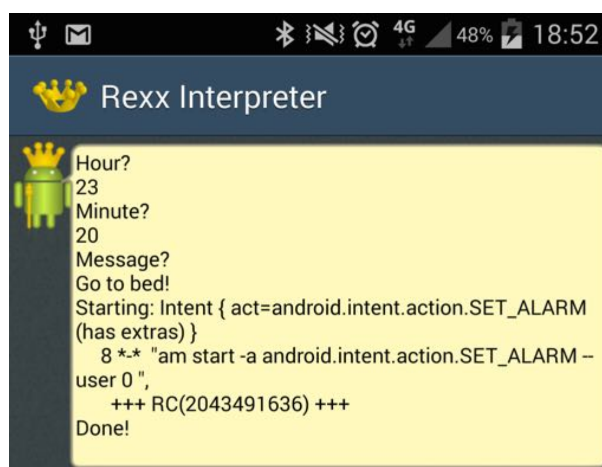


Figure 4: Alert creation - user interaction

After the alert is created, one can go to the alert menu and the newly created alert is displayed as in Figure 5.

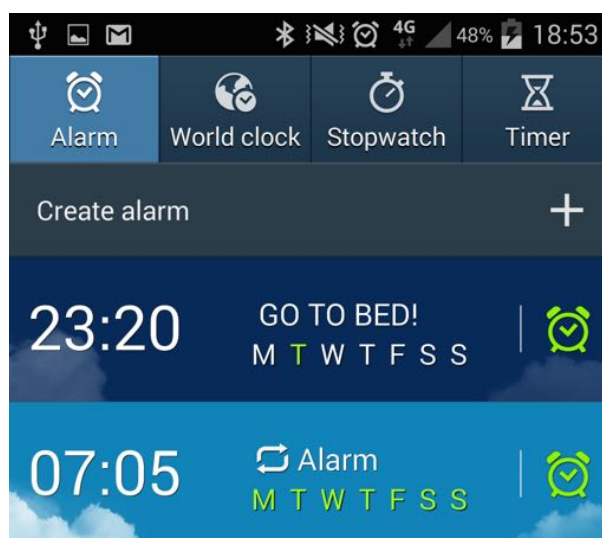


Figure 5: Alert created

3.3 Adding Calendar Events

This example illustrates how to create a simple new event for the calendar. The script does not require any special permissions.

```
NUMERIC DIGITS 18

SAY "Title?"
PULL title

SAY "Date? (e.g. 12 Feb 2012)"
PARSE PULL inputDate

SAY "Time? (e.g. 12:30)"
PULL inputTime
PARSE VAR inputTime inputHours ":" inputMinutes

SAY "Duration? (in hours)"
PULL durationHours

daysSinceEpoch = DATE("B", inputDate) - DATE("B", "01 Jan 1970")
millisSinceEpoch = daysSinceEpoch*24*60*60*1000

millisSinceEpoch = millisSinceEpoch + (inputHours*60+inputMinutes)*60*1000

millisSinceEpoch = millisSinceEpoch - 60*60*1000

"am start -a android.intent.action.INSERT --user 0 "          ,
    "-d Events.CONTENT_URI -t vnd.android.cursor.dir/event " ,
    "--es title '" title "'"                                  ,
    "--el beginTime " millisSinceEpoch                        ,
    "--el endTime " millisSinceEpoch+durationHours*60*60*1000
```

Listing 3: Create calendar event

To create events, the begin- and endtime must be provided in milliseconds since the Linux epoch (i.e. since the beginning of the Unix time, 01. January 1970). Therefore, some calculations have to be done before the activity to actually create the event can be started.

Line 1 tells the compiler to do calculations with a precision of 18 digits. As the default precision of 9 digits is insufficient (the numbers for the milliseconds can exceed 10^{12}, i.e. 13 digits), the used precision has to be increased. Figure 6 shows the milliseconds output without increased precision. The activity, which is started later, does not accept this notation. Line 3 to 14 ask the user for a title, the start date and time and the duration of the event. `PARSE PULL` is used to keep the input date case-sensitive (otherwise Rexx would transform the input to uppercase characters).

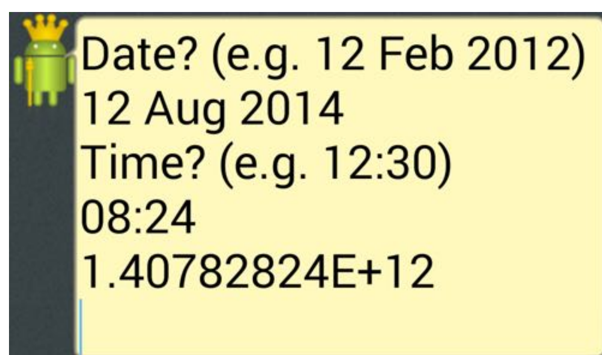


Figure 6: Calculating milliseconds without specified precision

After that, the milliseconds since the Linux epoch are calculated. Line 21 corrects the time difference between GMT and Austrian wintertime. For the sake of simplicity summertime is ignored, but therefore events created in the summertime span will have a one hour difference to the actual desired time.

Line 23 starts the activity and some additional parameters are defined:

- **-a android.intent.action.INSERT**: Defines the action to be executed.
- **--user 0**: Specifies the user (Without this, there will be an exception).
- **-d Events.CONTENT_URI**: Specifies the intent's data URI.
- **-t vnd.android.cursor.dir/event**: Specifies the intent's MIME type.
- **--es title title** : Specifies the title for the event. Here, the variable `title`, which holds the user's input, is used.
- **--el beginTime millisSinceEpoch** : Defines the start time for the event. Must be given in milliseconds since epoch (since 01. January 1970 00:00:00).
- **--el endTime millisSinceEpoch**: Defines the end time for the event. Must be given in milliseconds since epoch (since 01. January 1970 00:00:00).

Unfortunately, `Date("F")` and `Date("T")` do not work in Rexxoid, so milliseconds have to be calculated in a little more complex way, as in the example above. Figure 7 shows the error message that comes up if `Date("T")` is called.

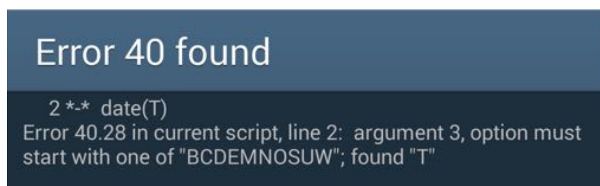


Figure 7: The date(T) error

Figure 8 shows the result of this example. The user wants to create a two hour long event on January 18, 2015 17:15 with the title "Great Event".

Figure 8: Event creation - user interaction

After the user enters the duration as shown in Figure 8, the calendar event creation screen (Figure 9) is displayed. After modifying the last settings, the event can finally be saved.

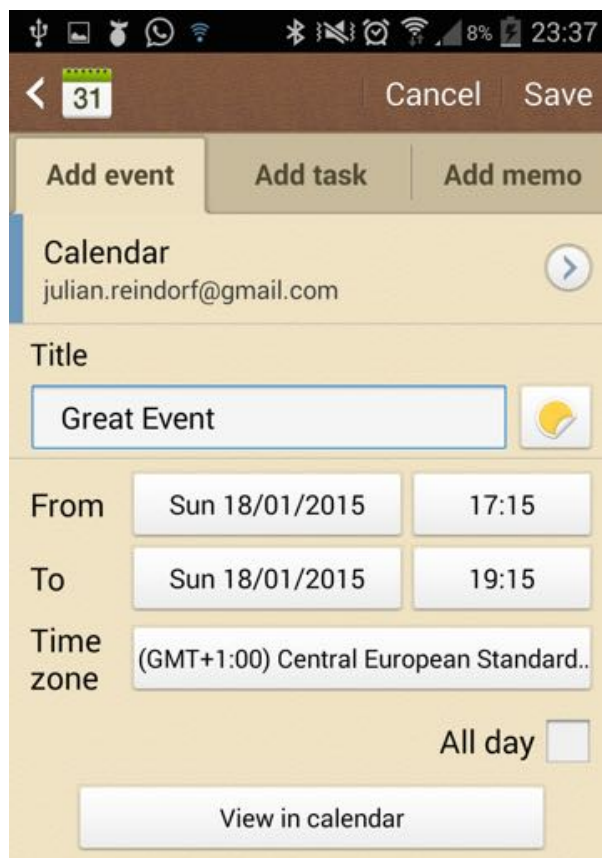


Figure 9: Event creation - settings

3.4 Command Line

This example lets the user enter commands which are then sent to the Android shell. The user can stop the script by typing 'stop' or 'STOP'.

```
1 DO WHILE TRANSLATE(input) <> 'stop'
2     PARSE PULL input
3     input
4 END
5 SAY "*Exited*"
6 EXIT
```

Listing 4: Send commands to Android shell

Some Linux commands that could be interesting to execute are:

- **ls**: Lists all files in the current directory.
- **ls -la**: Lists all files in the current directory (with more information than only `ls`).

- **ls /system/bin**: Shows all available shell programs. However, most of them require root user permissions to be executable.
- **uptime**: Shows the current uptime of the device.

In Figure 10 the user entered the "uptime" command. The command is forwarded to the Android shell and the result is displayed immediately.

Figure 10: Command line showing uptime

3.5 Creating Emails

This example shows how to create an email with a specified receiver, subject and text message.

```
1 text = ""
2 DO i=1 TO 100
3     text = text i
4 END
5 "am start -a android.intent.action.SEND --user 0 " ,
6     "-t 'text/plain' "                              ,
7     "-e to 'john.doe@gmail.com' "                   ,
8     "-e android.intent.extra.SUBJECT '1 to 100!' " ,
9     "-e android.intent.extra.TEXT '"text"'"
```

Listing 5: Create email

The purpose of lines 1-4 is to have some text in the variable `text`. The fifth line calls the action with the following settings:

- **-a android.intent.action.SEND**: Defines the action to be called.
- **--user 0**: Specifies the user (Without this, there will be an exception).
- **-t 'text/plain'**: Specifies the MIME-Type. This specifies it as plain text.
- **-e to 'john.doe@gmail.com'**: The extra string argument "to" specifies the receiver of the email.
- **-e android.intent.extra.SUBJECT '1 to 100!'**: Specifies "1 to 100!" to be the subject of the email.
- **-e android.intent.extra.TEXT text**: Specifies the value of the `text` variable to be the body of the email.

Note that the email is not sent by the script. It only opens the email application with preallocated fields.

This example includes no user interaction. Therefore, the result will always be the same, as shown in Figure 11. Note that the sender is not explicitly set in the script, instead the default sender is chosen.

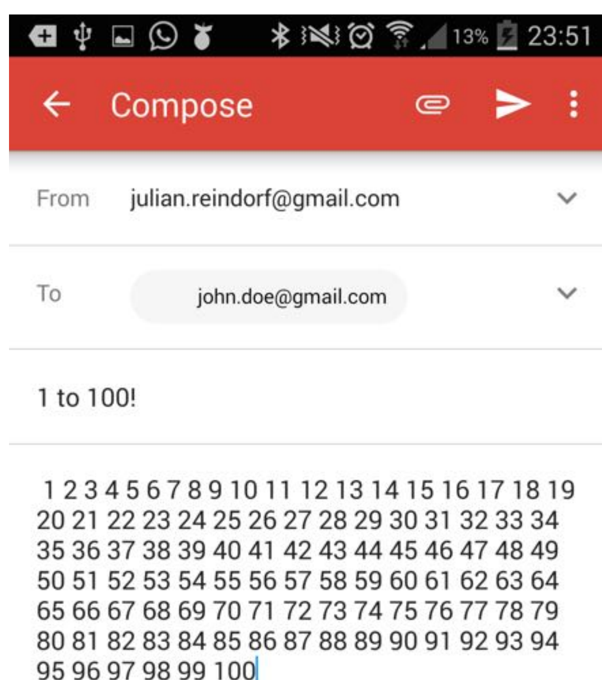


Figure 11: Rexxoid created email

3.6 Opening Google Maps

The following script opens the Google Maps application showing the requested location. The location can be almost anything like a street, city, country, water, mountain etc.

```
1 SAY "What do you want to see?"
2 input = LINEIN()
3 location=""
4 DO WHILE LENGTH(input) > 0
5     PARSE VAR input char +1 input
6     location=location|| "%"||C2X(char)
7 END
8 "am start -a android.intent.action.VIEW --user 0 -d 'geo:0,0?q="location"'"
```

Listing 6: Open Google Maps

The user has to enter the requested location. After that, every character of the entered location string is converted to its hexadezimal value, preceded by a percentage sign ("%"). This step is necessary, otherwise characters except for letters and numbers will not work correctly (e.g. blank (" ") or ampersand ("&")).

Line 9 starts the activity with the following settings:

- **-a android.intent.action.VIEW**: Defines the action to be called.
- **--user 0**: Specifies the user (Without this, there will be an exception).
- **-d 'geo:0,0?q="location"'**: Specifies the intent's data URI. In this case, the value of the variable `location` is set as the "q" parameter. In this example, the latitude (0) and longitude (0) have no effect.

Alterations of the intent used above can be:

- **-d 'geo:50,-70'** will show the map at the specified latitude (50) and longitude (-70).
- **-d 'geo:50,-70?z=10'** will show the map at the specified coordinates with the specified zoom (here, 10). The zoom level can be an integer between 1 (maximal zoom out) and 20 (maximal zoom in).

Note that it is not possible to use the "q" and the "z" parameter in the same activity call.

Figure 12 shows the script in action. As can be seen, the user wants to see Sydney.

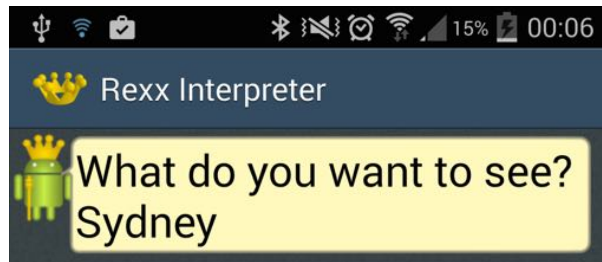


Figure 12: User chooses location

After the user choses the desired location, Google Maps starts and focuses on it. This can be seen in Figure 13.

Figure 13: Google Maps showing Sydney

3.7 Using the Speaker

This short example shows how to use the speakers to read out text.

```
1 CALL CHAROUT "Speaker:lang=en", "I am the voice of Rexxoid"
2 CALL CHAROUT "Speaker:lang=de", "Und ich kann auch andere Sprachen"
3 CALL CHAROUT "Speaker:lang=fr", "Au revoir"
```

Listing 7: Use speaker

The example consists of only one command (which is executed a few times). The first argument is an URI and defines the language for the reading aloud. It follows the pattern "Speaker:lang=<Code>", where <Code> is to be replaced by the two character language abbreviation. The second argument defines the text to be read. Of course, it is also possible to specify a different language in the URI and the text to be actually read aloud (e.g. "de" for German reading, and supplying an English text). This will, however, result in a hard to understand sound output.

3.8 Reading and Writing Files

This example shows how to write text into a file and retrieve the text afterwards from the file again. If the file does not exist, it is newly created.

```
1 path = "/sdcard/"
2 name = "foo.txt"
3 CALL CHAROUT path || name, "Written to file at" DATE() TIME()
4 CALL CHAROUT path || name, "Some interesting text"
5
6 text = CHARIN(path || name,1,1000)
7 SAY text
```

Listing 8: Write to and read from file

In the first two lines of the script, the path and the name of the file are defined. After that, in line 3 the file is created and some text, inluding date and time, are written into it. As can be seen, the first argument specifies the file path and name, while the second argument specifies the string to be written into the file. Line 4 adds more text into the existing file. Note that the file is not deleted and recreated, instead every further call of the charout method will append the specified string to it.

Line 6 retrieves the content from the previously created file and stores the string into the `text` variable. Again, the first parameter specifies the path and filename to be read.

The second parameter defines the index of the first character to be read out. The last parameter tells the script how many characters should be read. It is not mandatory to define the last two parameters, however if they are not supplied, only the first character of the file will be retrieved. In this example, reading starts with the first character of the file and includes up to 1000 characters (which is, in this case, the entire file). Line 7 finally prints the file's content.

Note that further executions of the script, without in between deletion of the created file, will append the specified strings, i.e. the file will not be deleted automatically and the printed text will include the text created in previous executions.

Figure 14 shows the outcome of the script. The text is first saved to a file "foo.txt" and, after that, readout and displayed.

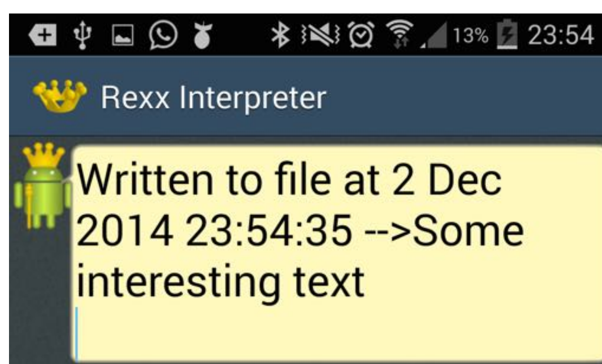


Figure 14: Text read from file

3.9 Sending Keyevents

The following example shows how to use keyevents.

```
1 DO 5
2     "input keyevent 24"
3     "sleep 0.6"
4 END
5 DO 5
6     "input keyevent 25"
7     "sleep 0.6"
8 END
```

Listing 9: Send keyevent

It can be seen that keyevents can easily be triggered by using "input keyevent <number>" as in line 2 and 6. The keyevents used in this example are 24 (increase volume)

and 25 (decrease volume). The events are fired 5 times each, having a delay of 0.6 seconds in between. The "sleep <seconds>" call uses the android sleep function, instead of Rexx's. This is due to the fact that the Rexx built in sleep function results in an error. Android's sleep function works well.

Keyevents can be almost anything, ranging from volume buttons, the home button, letters and numbers to the camera or the caps lock key. For a list of all keyevents visit `http://developer.android.com/reference/android/view/KeyEvent.html` [see Andr14c].

Figure 15 shows the script in action. The loops create and execute the input and the sleep function calls. In the upper half of the figure, a change in the volume can be seen (which is caused by the calls of the keyevent function).

Figure 15: Keyevent volume

3.10 Opening the Browser

This short example shows how to open an URL in a browser.

```
1 SAY "Which url would you like to visit?"
2 PARSE PULL url
3 "am start -a android.intent.action.VIEW --user 0 -d http:" || url
```

Listing 10: Open the browser

In line 1 and 2 the user is asked for the url to be visited. It is then stored in the `url` variable.

Line 3 starts the activity with the following settings:

- **-a android.intent.action.VIEW**: Defines the action to be called.
- **--user 0**: Specifies the user (Without this, there will be an exception).
- **-d http: || url**: Specifies the intent's data URI. The URI follows the following pattern: "http:<url>", where <url> is to be replaced with e.g. "www.wu.ac.at".

Figure 16 shows the script in action. As can be seen, the user chooses www.cnn.com to be displayed.

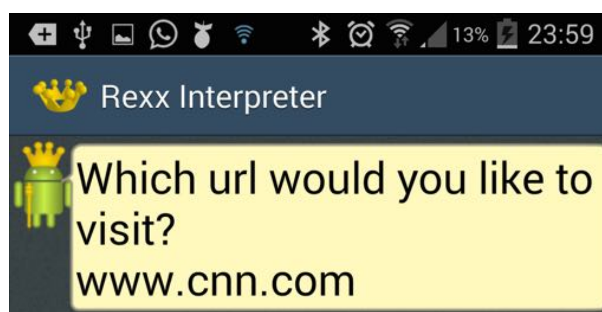


Figure 16: User chooses url

After the desired url is entered, the browser opens and the specified website is displayed (see Figure 17).

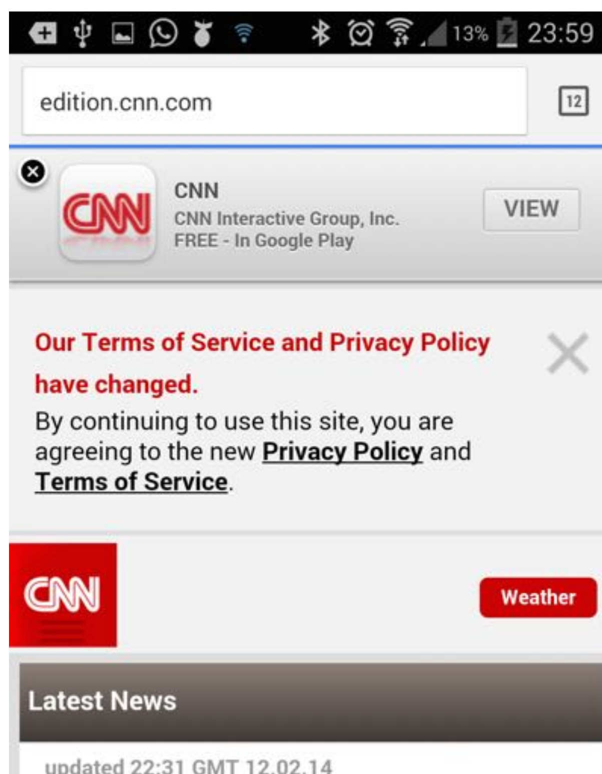


Figure 17: www.cnn.com

4 Limitations of Rexxoid

At the time of writing (fall 2014) there seems to be no documented Rexxoid script, which starts an activity and, after that, executes any further command. When trying to do so, the script crashes with the infamous message "Unfortunately, Rexx has stopped". To illustrate the issue the following example is provided.

```
"am start -a android.intent.action.VIEW --user 0 -d 'geo:0,0?q=vienna'"
"sleep 10"
"am start -a android.intent.action.VIEW --user 0 -d 'geo:0,0?z=16'"
```

Listing 11: Open Google Maps and zoom in

This should open Google Maps showing Vienna. Then, after a 10 seconds break, the zoom level should be changed to 16.

While the first two lines perform as expected, the script crashes at the third one. However, when sending these commands via the Android Debug Bridge (adb), they work well. This issue is also posted in the Google group comp.lang.rexx. By fall 2014, there is no solution to this.

Another limitation of Rexxoid is that text or any other result produced by the Android system cannot be retrieved and stored into a Rexx variable. Instead, it can only be displayed as output. This heavily reduces the number of use cases of Rexxoid.

Unfortunately, the Rexxoid community seems to be limited to a very small group of users. Probably, this is the result of other implementations (e.g. BRexx) of Rexx running on Android, which provide more features than simply executing Android shell commands. The small Rexxoid community also results in very few available Rexxoid script examples. Rexxoid comes with some examples preinstalled, but only one of them actually includes a call to the Android shell.

Note that the author did not reach Rexxoid's author to clarify if these limitations are actually results of errors or incompletions of Rexxoid or if there are simply no documented examples for these issues.

5 Rexxoid versus BRexx

Rexxoid and BRexx are both interpreters of Rexx for Android. While BRexx builds on the scripting layer for Android (SL4A), Rexxoid does not make use of it. As only BRexx uses this functionality, SL4A is considered to be a part of BRexx for the comparison. In this section they will be compared and major differences highlighted.

Both applications were tested on the same device, Samsung Galaxy S4 (model number GT-I9505) running Android version 4.4.2. Therefore there should be no differences owing to the used device. All statements refer to the time of writing, which is fall 2014.

This section was written in cooperation with Eva Gerger [see Gerg14].

5.1 Comparison

Installation The Rexxoid application can be downloaded directly from the Google Play store. Android devices install the downloaded application automatically and after that the installation process is finished. BRexx needs two applications, SL4A and BRexx itself, neither of them can be downloaded from the Google Play store. They have to be obtained from external sources. This requires also the permission to install applications from third parties, which is not necessary for Rexxoid.

Example repository Rexxoid comes with a few preinstalled examples. Of these few examples only one uses Android functionality. Also there are no further examples for Rexxoid on the web which use Android functionality. In comparison, BRexx also comes with only a few preinstalled examples. However, some more can be found on the web. Admittedly, those are example scripts in other languages (e.g. in Python or JavaScript), but as they also use SL4A functions they can be adapted quite easily.

Functionality The functionality of Rexxoid is very limited. Besides "normal" Rexx code, only Android shell commands can be used. Hardly any predefined functions are available. BRexx, on the other hand, offers a great variety of functions. Almost any Android component can be addressed in BRexx (e.g. sensors can be utilised). Additionally, in contrast to Rexxoid, BRexx scripts can run in the background, which leads to even further fields of application.

Although both applications come with a small set of granted Android permissions, for some functions additional permission are necessary. Adding permissions is rather

effortful in both cases, it can only be done by editing the source code of the application (specifically, the Manifest file) and therefore requires a reinstallation. BRexx has the advantage that written scripts are not deleted in this process, while Rexxoid's need to be saved in advance.

Usability Rexxoid's user interface is very slim and the navigation is rather inconvenient (e.g. the script has to be opened in editing mode before it can be executed). BRexx's interface is more sophisticated and offers handy options (e.g. Save & Run). Also the menus are very well structured and useful. This makes getting started very easy.

Performance Both applications come with the same script to measure their performance (Average REXX clauses-per-seconds by Mike Cowlishaw). When executing the script, Rexxoid ends up with about 450.000 clauses per second, whereas BRexx reaches about 1.1 million. It can be seen that BRexx is about twice as fast as Rexxoid.

Community Rexxoid's community consists only of a handful of people. BRexx's community is actually also very little, but because of the big SL4A community support for many of the upcoming problems can be found in this community as well.

Documentation There is no documentation of Rexxoid. BRexx does not offer any documentation as well. But at least SL4A offers a list of available functions and their arguments. Unfortunately, there is no information on dependencies of the functions (i.e. function x has to be called before function y or y will not work). Nevertheless, the built in API browser is comfortable to use.

Readability Reading and understanding Rexxoid scripts is quite difficult. Android shell commands are very long and not self-explanatory. Due to very meaningful function naming in SL4A, BRexx scripts are easy to read and understand.

Debugging Debugging in Rexxoid is easier than in BRexx. Exceptions thrown by the operating system or by Rexxoid itself are displayed in Rexxoid's default output console. Syntactical errors are noticed immediately and prevent execution of the script. BRexx does not display any errors by default. After execution of a script, the unfiltered logcat output can be displayed (which is really long and not overseeable). So the best way

of displaying errors and debugging is to connect the device to a computer and use the Android Debug Bridge to display a filtered logcat output.

Software updates The newest version of Rexxoid is from September 2014, while BRexx's newest version is from March 2013. Rexxoid is being developed continiously, BRexx has longer release cycles.

5.2 Recommendation

Table 2 summarises the results of section 5.1. The application which performs better in a specific aspect is marked with a plus sign (+), the less well performing application marked with a minus sign (–). If they are equal, a tilde is used (~).

Aspect	Rexxoid	BRexx
Installation	+	–
Example repository	–	+
Functionality	–	+
Usability	–	+
Performance	–	+
Community	–	+
Documentation	–	+
Readability	–	+
Debugging	+	–
Software updates	~	~

Table 2: Comparison of Rexxoid and BRexx

As Table 2 indicates, BRexx is the better choice in almost any perspective. Maybe Rexxoid improves and enriches its feature set in the future, but currently the clear recommendation is BRexx.

6 Conclusion

Rexxoid enables developers to write short Rexx scripts for Android devices easily. However, if Android functionality should be used as well, it becomes complex and the functionality is very limited. Unfortunately, Rexxoid can not run in the background, which makes it unusable for many tasks.

The Rexxoid community is really small, which makes it even harder to find support. There are almost no nutshell examples available on the web. Additionally, hardly any scripts are preinstalled so it is tricky to get started with Rexxoid.

What makes it even more difficult to write scripts is the fact that there is not a single line of documentation available. Of course, there is a documentation for Rexx itself, but not for the interaction with Android.

The other approach to run Rexx on Android discussed in this paper, BRexx, seems more practical, comes with a greater community and a documentation. Furthermore, BRexx includes features that are absolutely needed in Rexxoid, especially scripts running in the background.

Maybe the nutshell examples in this paper help someone to fix a problem or develop new scripts. At the current stage, one should definitely consider using an alternative.

References

[Andr14a] Android.: Android Debug Bridge. http://developer.android.com/tools/help/adb.html, Accessed on 2014-12-23.

[Andr14b] Android.: Installing the Android SDK. http://developer.android.com/sdk/installing/index.html, Accessed on 2014-12-23.

[Andr14c] Android.: KeyEvent. http://developer.android.com/reference/android/view/KeyEvent.html, Accessed on 2014-12-23.

[Butl11] Butler, Margaret: Android: Changing the Mobile Landscape. In: IEEE Pervasive Computing 10 (2011) 1, p. 4-7.

[Flat13] Flatscher, Rony G.: Introduction to REXX and ooRexx. Facultas, 2013.

[Gerg14] Gerger, Eva.: BRexx: Running Rexx on Android Systems. Vienna University of Economics and Business, Seminar Thesis, 2014.

[Goog14] Google Play.: Rexx for Android. https://play.google.com/store/apps/details?id=com.jaxo.android.rexx, Accessed on 2014-12-23.

[Inte14] International Data Corporation.: Smartphone OS Market Share, Q3 2014. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, Accessed on 2014-12-06.

[Open14] Open Handset Alliance.: Industry Leaders Announce Open Platform for Mobile Devices. http://www.openhandsetalliance.com/press_110507.html, Accessed on 2014-12-06.

[Wiki14] Wikipedia.: Intent (Android). http://en.wikipedia.org/w/index.php?title=Intent_(Android)&oldid=626240352, Accessed on 2014-12-23.

REXXOID

Running Rexx on Android

Julian Reindorf

26th International Rexx Symposium, 2015

CONTENTS

- Rexxoid introduction
- Nutshell examples
- Limitations

REXXOID

- Available on Google Play Store
- Execute Rexx Code
- Send commands to Android shell
- Read out aloud
- No documentation

HELLO WORLD

```
1 SAY "Hello World";
```

OPENING THE BROWSER

```
1 SAY "Which url would you like to visit?"
2 PARSE PULL url
3 "am start -a android.intent.action.VIEW
  --user 0
  -d http:" || url
```

SENDING KEYEVENTS

```
1 DO 5
2   "input keyevent 24"
3   "sleep 0.6"
4 END
5 DO 5
6   "input keyevent 25"
7   "sleep 0.6"
8 END
```

COMMAND LINE

```
1 DO WHILE TRANSLATE(input) <> 'stop'
2   PARSE PULL input
3   input
4 END
5 SAY "*Exited*"
6 EXIT
```

USING THE SPEAKER

```
1 CALL CHAROUT "Speaker:lang=en", "I am the voice of Rexxoid"
2 CALL CHAROUT "Speaker:lang=de", "Und ich kann auch andere
  Sprachen, wie Deutsch"
3 CALL CHAROUT "Speaker:lang=fr", "Au revoir"
```

READING AND WRITING FILES

```
1 path = "/sdcard/"
2 name = "foo.txt"
3 CALL CHAROUT path || name, "Written to file at" DATE() TIME()
4 CALL CHAROUT path || name, "Some interesting text"
6 text = CHARIN(path || name,1,1000)
7 SAY text
```

CREATING EMAILS

```
1 text = ""
2 DO i=1 TO 100
3   text = text i
4 END
5 "am start -a android.intent.action.SEND
--user 0
-t 'text/plain'
-e to 'john.doe@gmail.com'
-e android.intent.extra.SUBJECT '1 to 100!'
-e android.intent.extra.TEXT '"text"'"
```

OPENING GOOGLE MAPS

```
1 SAY "What do you want to see?"
2 input = LINEIN()
3 location=""
4 DO WHILE LENGTH(input) > 0
5   PARSE VAR input char +1 input
6   location=location|| "%"||C2X(char)
7 END
8 "am start -a android.intent.action.VIEW
  --user 0
  -d 'geo:0,0?q="location"'"
```

SETTING ALERTS

```
1 SAY "*Alert Creation*"
2 SAY "Hour?"
3 PULL hour
4 SAY "Minute?"
5 PULL minute
6 SAY "Message?"
7 PULL message
8 "am start -a android.intent.action.SET_ALARM
  --user 0
  --ei android.intent.extra.alarm.HOUR " hour "
  --ei android.intent.extra.alarm.MINUTES " minute "
  -e android.intent.extra.alarm.MESSAGE '" message "'
  --ez android.intent.extra.alarm.VIBRATE false
  --ez android.intent.extra.alarm.SKIP_UI true"
9 SAY "Done!"
```

ADDING CALENDAR EVENTS

```
1 NUMERIC DIGITS 18
2 SAY "Title?"
3 PULL title
4 SAY "Date? (e.g. 12 Feb 2012)"
5 PARSE PULL inputDate
6 SAY "Time? (e.g. 12:30)"
7 PULL inputTime
8 PARSE VAR inputTime inputHours ":" inputMinutes
9 SAY "Duration? (in hours)"
10 PULL durationHours
11 daysSinceEpoch = DATE("B", inputDate) - DATE("B", "01 Jan 1970")
12 millisSinceEpoch = daysSinceEpoch*24*60*60*1000
13 millisSinceEpoch = millisSinceEpoch + (inputHours*60+inputMinutes)*60*1000
14 millisSinceEpoch = millisSinceEpoch - 60*60*1000
15 "am start -a android.intent.action.INSERT
   --user 0
   -d Events.CONTENT_URI
   -t vnd.android.cursor.dir/event
   --es title '" title "'
   --el beginTime " millisSinceEpoch "
   --el endTime " millisSinceEpoch+durationHours*60*60*1000
```

LIMITATIONS

- Execute commands after starting an activity

```
1 "am start -a android.intent.action.VIEW
  --user 0
  -d 'geo:0,0?q=vienna'"
2 "sleep 10"
3 "am start -a android.intent.action.VIEW
  --user 0
  -d 'geo:0,0?z=16'"
```

Thanks for your attention!

Enjoy the rest of the symposium :)

julian.reindorf@gmail.com

Vienna University of Economics and Business

Institute for Management Information Systems

Course 0208, Projektseminar aus Wirtschaftsinformatik (Schiseminar)

BRexx: Running Rexx on Android Systems

Eva Gerger, 1151555

Supervisor: ao. Univ.Prof. Mag. Dr. Rony G. Flatscher

December 25, 2014

Declaration of Authorship

"I do solemnly declare that I have written the presented research thesis by myself without undue help from a second person others and without using such tools other than that specified.

Where I have used thoughts from external sources, directly or indirectly, published or unpublished, this is always clearly attributed.

Furthermore, I certify that this research thesis or any part of it has not been previously submitted for a degree or any other qualification at the Vienna University of Economics and Business or any other institution in Austria or abroad."

Date:

Signature:

Contents

1 Introduction **1**

1.1 Rexx . . . 1

1.2 Android . . . 2

1.2.1 Overview . . . 2

1.2.2 Android Applications . . . 3

1.2.3 Functional Principle . . . 3

1.3 BRexx . . . 4

1.3.1 Installation . . . 4

1.3.2 Debugging . . . 4

1.3.3 Application . . . 5

2 Nutshell Examples **6**

2.1 Hello World - dlroW olleH . . . 6

2.2 Vampire . . . 9

2.3 SMS to.. . . . 11

2.4 How is My Battery Doing? . . . 14

2.5 Hi, Bluetooth! . . . 16

2.6 Look @ Maps . . . 21

2.7 Phone-info . . . 23

2.8 Scan My Barcode . . . 26

3 BRexx versus Rexxoid **27**

3.1 Comparison . . . 27

3.2 Recommendation . . . 29

4 Conclusion **30**

List of Figures

1 List of scripts 5
2 Script editing mode 5
3 Running the Hello World script 8
4 Toast and commandline output 9
5 Vampire - Result screen 11
6 SMS - Dialog box - Number of SMS to send 12
7 SMS - Dialog box - Phone number 13
8 SMS - Dialog box - Received SMS 14
9 Battery status - Output 15
10 Bluetooth permission request 17
11 Bluetooth server - Information messages 18
12 Bluetooth server - Message 19
13 Bluetooth client - Received message 20
14 Google Maps showing WU Vienna 23
15 Phone-info 25

List of Tables

1 Comparison of Rexxoid and BRexx 29

Listings

1 Hello World - dlroW olleH 6
2 Vampire 10
3 SMS 12
4 How is my battery doing? 14
5 Bluetooth Server 16
6 Bluetooth Client 16
7 Look @ Maps 21
8 Phone-info 24
9 Scan my Barcode 26

Abstract

Mobile devices are on the rise, most of them using the Android operating system. However, developing applications for Android is complex and hard to learn. Scripting Layer for Android in combination with BRexx (an interpreter for the programming language Rexx) is able to provide relief for many problems. This work provides nutshell examples for BRexx. A short comparison of BRexx and Rexxoid, another Android Rexx interpreter, is given. The advantages of BRexx over Rexxoid are highlighted.

1 Introduction

The aim of this work is to run Rexx on Android systems. Therefore the Scripting Layer for Android (SL4A), used to make scripting languages available to Android, will be used to be able to run Rexx scripts directly on an Android device. The main focus is on the development of nutshell examples running Rexx on Android.

First, a short overview of Rexx, Android and BRexx will be given. Next, the set-up of the necessary environment will be explained briefly. This will be followed by the implementation and description of different nutshell examples. A comparison to Rexxoid, a different interpreter available for Android, is made as well. Last, the conclusion summarises findings and problems.

1.1 Rexx

Rexx is a programming language which has been developed in 1979 for IBM mainframes because it is easier to read and understand than many other programming languages. Due to the fact that various non-IBM implementations of Rexx exist, it can be seen that the Rexx language has a high impact. There is even an ANSI/INCITS standard defining the programming language Rexx. In 1997 IBM first released and distributed *Object Rexx*, a version of Rexx, which had been extended with object-oriented features. The source code of *Object Rexx* has been handed over to the special interest group "Rexx Language Association" in 2004, which then published *ooRexx - Open Object Rexx 3.0*, the first open source version of IBM's *Object Rexx*. ooRexx is an object oriented version of the Rexx programming language, which is not only fast and powerful, but also offers a great documentation. It is free, open source, runs on multiple platforms and can be further extended as well. [see Flat13, iii f.]

Rexx has some fundamental language concepts. The structure of the syntax and also

names of keywords are aimed at being easily readable as well as understandable. Although case can be used, Rexx does not distinguish between lower and upper case. Rexx is not strongly typed. Basically, everything is a string. Because of the importance of strings in Rexx, there are various string manipulation operators available. Also, there is no declaration mechanism in Rexx. Another focus is on staying adaptable, which is why Rexx does not reserve any key words. Rexx is meant to be kept small, so new features are only added if they offer a significant benefit for users. Those are also some of the major reasons that helped Rexx being widely used. [see Flat13, 24 ff.]

1.2 Android

Android is the most widely used mobile operating system. It dominates the global smartphone market with about 85% marketshare in the third quarter of 2014. Its biggest rival Apple owns only 11.7% of the market. [see Inte14]

Android first was developed by the Californian company Android Inc., which has later been bought by Google. Android is an open source project of the Open Handset Alliance. It consists of many firms, ranging from mobile operators, handset manufacturers, semiconductor companies and software companies to commercialisation companies and is managed by Google. The Open Handset Alliance is aimed at continuous innovation and openness, building enhanced experiences on mobile devices for users. [see Open14]

1.2.1 Overview

Android is a platform for developing and running of applications on mobile devices like smartphones, tablets, wearables and a lot more. But it is not only an operating system, it also includes middleware and a vast amount of mobile applications as well as a set of API libraries. The open development environment, which is build upon an open-source Linux kernel, is one of Android's key factors. Also, in contrast to other systems, there is no differentiation between applications developed by third parties and applications directly belonging to the system. They all use the same runtime environment and are developed using the same APIs. Therefore they all have the same significance. This also enables users and developers to replace applications originally belonging to the system with third party applications. [see Meie14, 27 ff.]

1.2.2 Android Applications

The basic building blocks of an Android application are Java-programmes, resources and the Manifest file. The **Java-programmes** are responsible for the (main) functionality of the application. Apart from that, they are the ones communicating with the user on the one hand and with the system itself on the other hand. **Resources** are various kinds of unalterable data like pictures, media etc. which are required by the application but not part of the program itself. The can either have the form of data of a specific type or the form of XML files, which can be used for layouts, strings and a lot more. The **Manifest file** declares all components of an application and some other properties, like permissions or hardware requirements, as well. The information that the development environment needs to create the application, the operating system needs for installing and running the application as well data required by Google Play Store to publish the application can be found in this file. [see Stau13, 13 ff.]

1.2.3 Functional Principle

Mobile applications have some special requirements that regular applications do not have. Physical resources are very limited. Different applications with different sources are working together closely, but still they need to immediately react to certain events like incoming calls. So, to enable smooth operation, applications have to be developed and run in a certain way. [see Stau13, 15 ff.]

Every application is run by a separate Linux user in a separate Sandbox, which basically is an isolated environment in which only the permissions required to perform the application's tasks are granted. The Android-environment enables different applications to interact with each other. Without this environment the interaction would not be possible, because the separation is very strict. To be able to use security relevant components or other applications, an Android application needs to have the regarding permissions, which it does not have by default. Permissions need to be declared in the Manifest file mentioned above. [see Stau13, 15 ff.]

Android applications are build upon a component model. Android components are special kinds of Java-classes, which are the main building blocks for the functionality of the application. Communication with these components is made via the use of platform-functions. Thus components of an application are only loosely coupled and can easily interact with components of other applications or can even be replaced by them. Android components are *Activities*, *Services*, *Broadcast-Receivers*, *Content-Providers* and *Applications*. The communication between *Activities*, *Services* and *Broadcast-Receivers* happens via specific messages, so called Intents. All the components have

to be declared in the Manifest file. [see Stau13, 15 ff.]

1.3 BRexx

BRexx makes Rexx available to the Scripting Layer for Android application. Therefore, Rexx scripts can be written and executed directly on Android devices, making Androids functionality and resources available to use.

1.3.1 Installation

In order to be able to run Rexx on Android, two applications have to be installed. The first one is *SL4A*, the Scripting Layer for Android [see Goog14]. The second necessary application is *BRexx.apk* [see Brex14]. After those two applications have been installed, the programming part is about to start. Both applications come with preinstalled examples, which can be used as a starting point.

1.3.2 Debugging

Debugging is a vital element of programming. In the BRexx application itself, debugging is very inconvenient and not properly working. It is only possible to view the unfiltered log output of Android (logcat). Unfortunately, only the last few lines can be seen, since scrolling is not working. Therefore, a better way is to attach the device with an USB-cable to a computer and make use of the Android Debug Bridge (adb). Executing `adb logcat | grep -i "SL4A"` provides proper messages.

1.3.3 Application

After starting the SL4A application, the user has to navigate to the Android folder. The content of the Android folder is a list of all Android - Rexx scripts. When tapping on one of the scripts, a small menu appears like in Figure 1. This allows the user to choose whether the script should be run, edited, saved or deleted. In case the user chooses to edit the script, it opens in a different screen, as shown in Figure 2. While editing a script, the user is presented with an additional menu after pressing the menu button. This menu allows the user to browse functions in the API browser, which shows all available functions and their arguments. Also, the script can be saved and run as well.

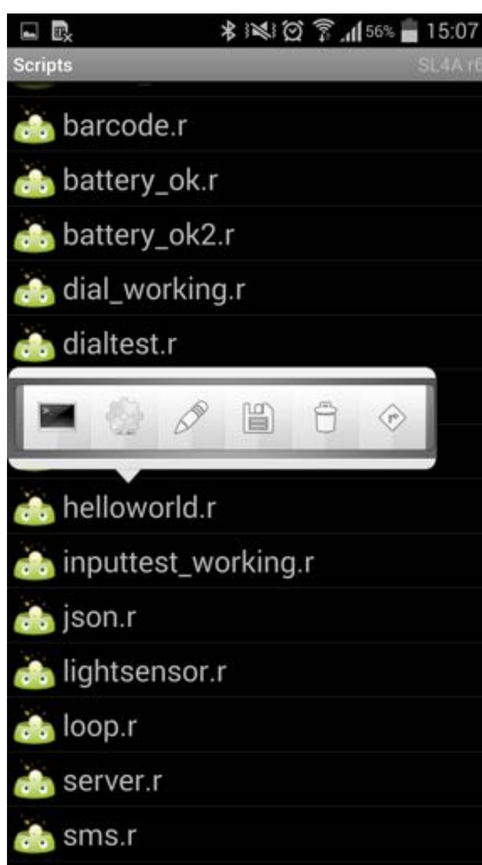


Figure 1: List of scripts

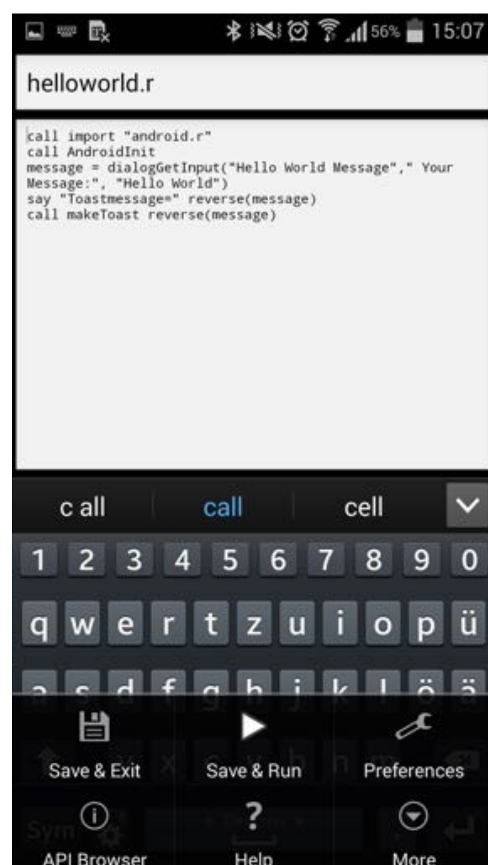


Figure 2: Script editing mode

2 Nutshell Examples

The following sections will demonstrate some of the useful functionality on short nutshell examples. The code as well as the results will be explained. Important functions and their parameters will be discussed as well. Some of the following examples are derived from examples written in other scripting languages on *https://code.google.com/p/android-scripting/wiki/Tutorials*.

2.1 Hello World - dlroW olleH

Traditionally the first example will be a short *Hello World* script. This means that the script simply prints Hello World on the command line or any other output media. In this case, the script will not only print Hello World. Before it does this, it applies one of Rexx's string functions, namely the `reverse` function.

```
1 call import "android.r"
2 call AndroidInit
3 message = dialogGetInput("Hello World Message"," Your Message:", "Hello World")
4 say "Toastmessage=" reverse(message)
5 call makeToast reverse(message)
```

Listing 1: Hello World - dlroW olleH

The script code can be seen in Listing 1. The first two lines of code provide the necessary initialisation to be able to use Android functionality.

Then the user is prompted for an input string. This is accomplished using the `dialogGetInput` function. This function has several input parameters, which are all optional.
Functionality: This function displays an input box which can be customised by the user. The box already has an OK and a Cancel button. The value that is entered is returned to script after the OK button has been pressed.
Parameters:

- **String title**: This is the title of the input box.
- **String message**: This is the message which will be displayed above the input box. It also has a default value, so in case that no message has been entered, the string "Please enter value" will be used.
- **String defaultText**: This is the default text that will be displayed in the input box before the user enters anything.

- **Return value**: This is the string that will be returned to the script that called the function.

After the user entered a string and pressed ok, this string is being reversed and printed in the SL4A application running script's command line environment. Also, a Toast is made using the reverse function and the input string.

For this the `makeToast` function is used.
Functionality: The string message is used to display a toast message for a short time.
Parameters: This function has only one input parameter.

- **String message**: This is the string that will be used to make the toast.

As it can be seen in Figure 3, the title of the input box is "Hello World Message". The message of the input box is "Your message:" and the defaultText is "Hello World".

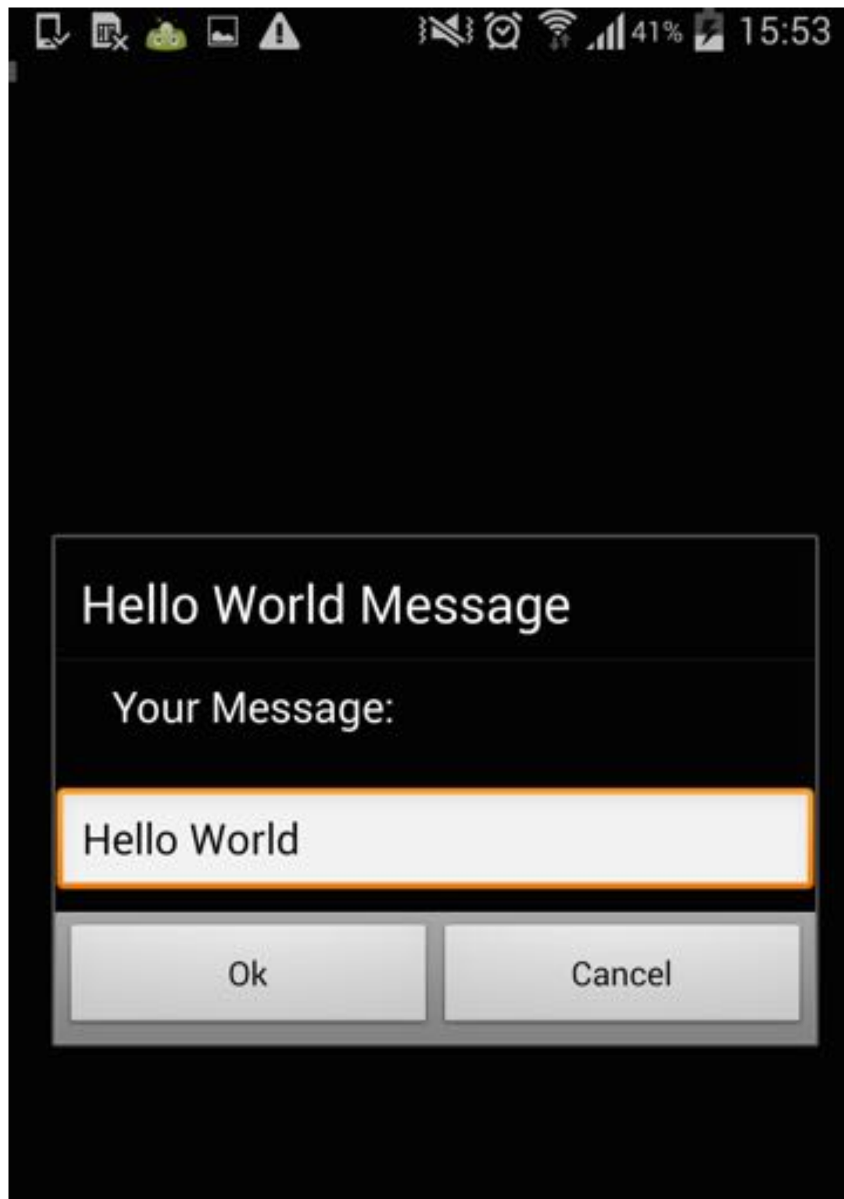


Figure 3: Running the Hello World script

The message used to create the toast is the reverse of Hello World, that is dlroW olleH, which can be seen in Figure 4.

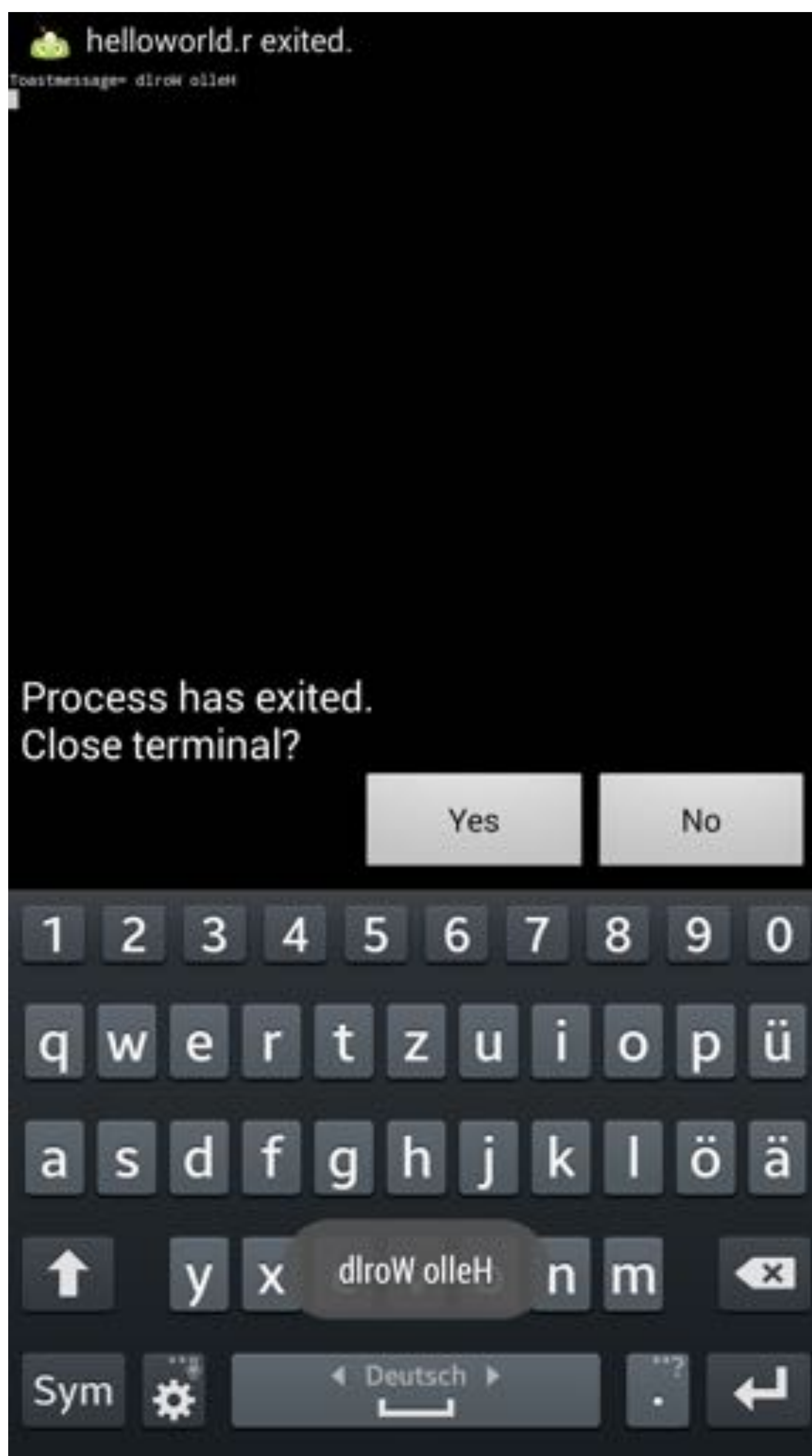


Figure 4: Toast and commandline output

2.2 Vampire

It is commonly known that vampires do not like (day-) light. This example simulates a vampire, who therefore also does not like light. The script uses the phone's light sensor to determine the current light value of the environment. In case the light value is above

a certain threshold, the phone activates the vibration mode. See Listing 2 for the script code.

```
1  call import "android.r"
2  call AndroidInit
3  call startSensingTimed 4, 500
4  call eventWaitFor "sensors"
5  do 10
6    strength = sensorsGetLight()
7    say strength
8    if strength > 100 then call vibrate 500
9    call sleep 4
10 end
11 say "this is the end"
```

Listing 2: Vampire

The first two lines are again used to import the `"android.r"` file and perform the necessary initialisation.

Then, sensing is started using the `startSensingTimed` function.
Functionality: This function starts the monitoring of sensor data which then can be queried.
Parameters: The function takes two parameters, both are required.

- **Integer sensorNumber**: This is the number of that sensor whose data should be recorded. Possible Values are 1 (All sensors), 2 (Accelerometer), 3 (Magnetometer) and 4 (Light).
- **Integer delayTime**: This refers to the time that should at least pass between two sensor readings and is measured in milliseconds.

Next the `eventWaitFor` function is called, which blocks the execution of the following code lines until a sensor event is received.
Functionality: This function blocks the execution of further code until a specified event is received.
Parameters: The function takes two parameters.

- **String eventName**: This is the event that is waited for, which always must be supplied.
- **Integer timeout**: This refers to the maximum time to wait, measured in milliseconds.

Subsequently, a loop is repeated 10 times. In this loop, the light value of the measurement is obtained using the `sensorsGetLight` function, which returns the last received light value. If this value is above 100, the vibration mode is activated for 500 milliseconds. If the value is below 100, nothing happens. The next line forces the script to wait for 4 seconds, using Rexx's `sleep` function which has only one parameter, i.e. the duration in seconds. After finishing the whole loop, the final message "this is the end" is printed. The final output screen can be seen in Figure 5.

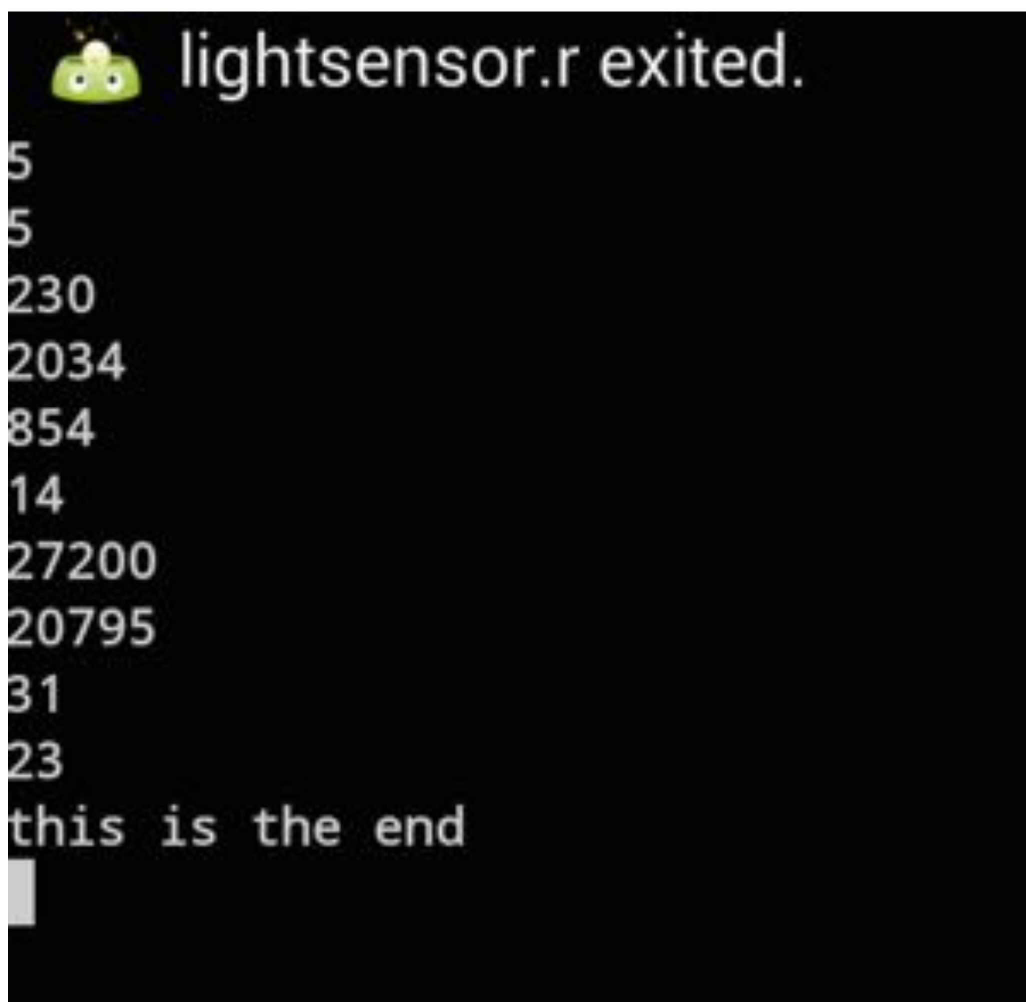


Figure 5: Vampire - Result screen

The numbers in Figure 5 are the light values. Every time the loop is passed through, the current value is printed. The final message can be seen on the last text line.

2.3 SMS to..

This nutshell example is intended to send SMS to multiple phone numbers. First, the user is asked how many messages will be sent. Then, a phone number is queried and an SMS, containing an ascending number, is sent. This happens for as many times as

the user initially specified. For this, see Listing 3.

```
call import "android.r"
call AndroidInit
x = dialogGetInput("Number of SMS to send","Please enter number of SMS to send:")
do i = 1 to x
  nr = dialogGetInput("Phone number","Please enter phone number:")
  call smsSend "tel:"nr, "You are number "i
end
```

Listing 3: SMS

The first two lines of the script are the same as in the previous scripts. The next line calls the `dialogGetInput` function, which is described in Section 2.1 in detail. The user has to enter to how many numbers SMS will be sent. The according dialog box can be seen in Figure 6.

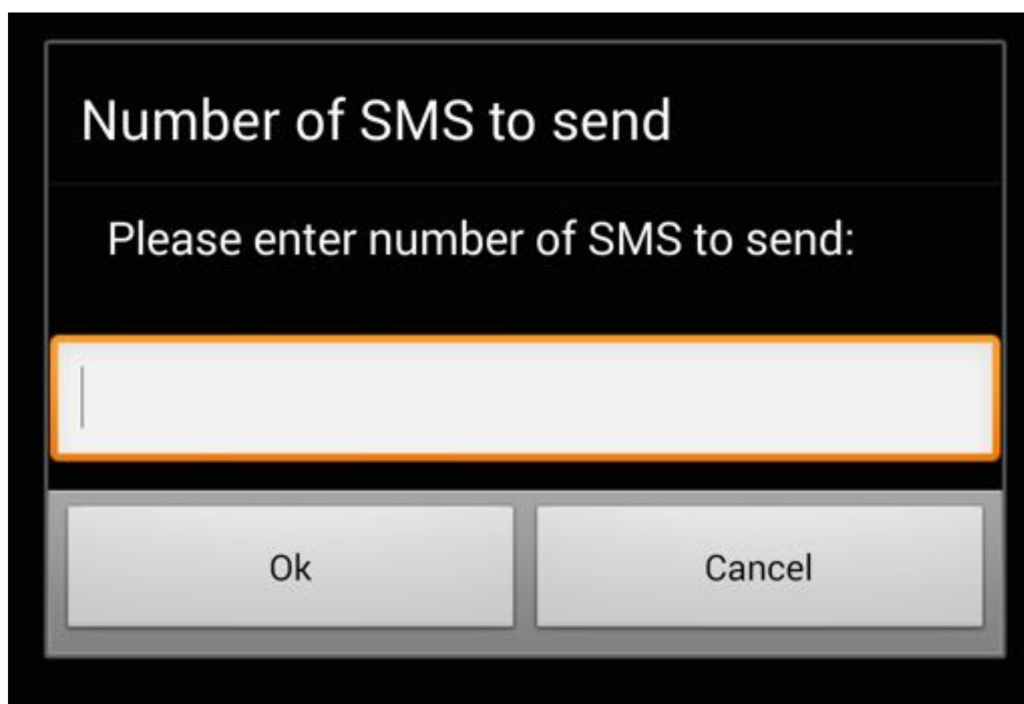


Figure 6: SMS - Dialog box - Number of SMS to send

Then, the following loop is executed as many times as the user has entered. In the loop, the `dialogGetInput` function is called again. This time it queries the user for a phone number, as it can be seen in Figure 7.

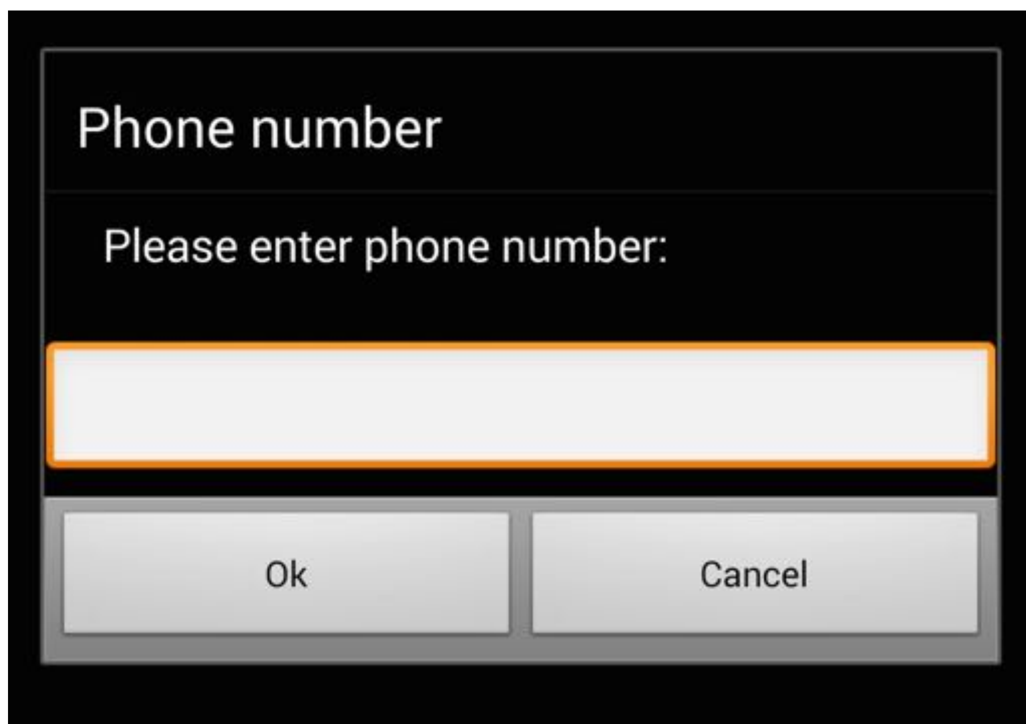


Figure 7: SMS - Dialog box - Phone number

An SMS containing the text "You are number" followed by an ascending number (starting from 1) is sent to the previously entered number.

This is achieved by using the `smsSend` function.
Functionality: This function sends an SMS to a specified number.
Parameters: This function requires two parameters, they cannot be omitted.

- **String destinationAdress**: This is the destination address of the message, typically this is a phone number.
- **String text**: This is the text of the message.

Now the sms looks like in Figure 8. Note that the SMS will not be listed twice on the receivers phone, this is owing to the fact that in this example the receiver is also the sender. The script is finished after all SMS have been sent.

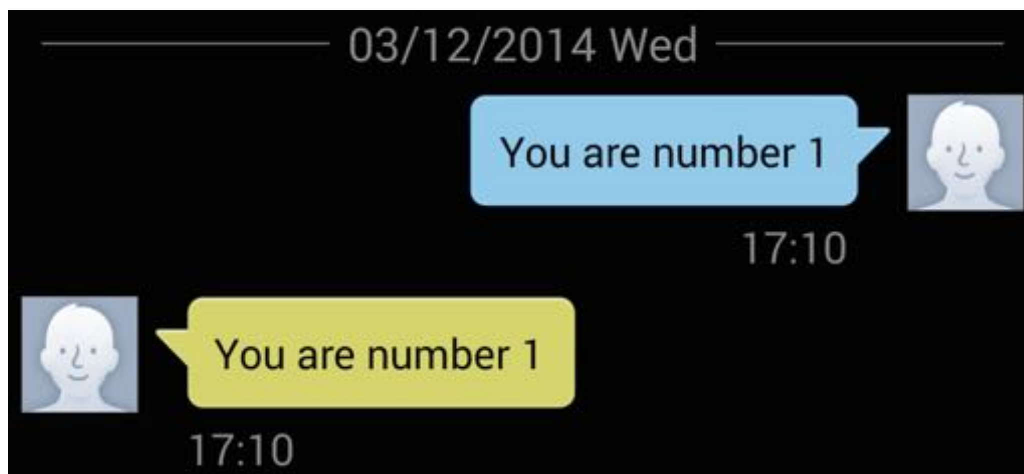


Figure 8: SMS - Dialog box - Received SMS

2.4 How is My Battery Doing?

The following example script demonstrates the use of functions around the phone's battery. The script code can be found in Listing 4. The script monitors the battery and notifies the user about how charged the phone's battery is.

```
1 call import "android.r"
2 call AndroidInit
3 call batteryStartMonitoring
4 call eventWaitFor "battery", 5000
5 a = batteryGetLevel()
6 say a
7 call batteryStopMonitoring
8 call makeToast Batterylevel a
```

Listing 4: How is my battery doing?

Again, the first two lines of code are needed for initialisation.

Next up is the call of the `batteryStartMonitoring` function.
Functionality: The call of this function starts tracking of the battery state. Also, *"battery"* events are generated by this function.
Parameters: This function does not receive any parameters.

Next, the `eventWaitFor` function is called to make the system wait for a battery event, at most for 5000 milliseconds. The function is explained in detail in Section 2. This is required, because the `batteryStartMonitoring` function takes some time before it is ready to operate.

The `batteryGetLevel` function is called in the following step.
Functionality: This function returns the latest received battery level in percent.
Parameters: This function does not have any input parameters.

The received value is then printed on the command line box of the application.

After that, the `batteryStopMonitoring` function is called.
Functionality: This function stops tracking of the battery state.
Parameters: This function does not have any input parameters.

In a final step the `makeToast` function is used to display the result also as Toast message. A detailed explanation of the `makeToast` function can be found in Section 2.1.

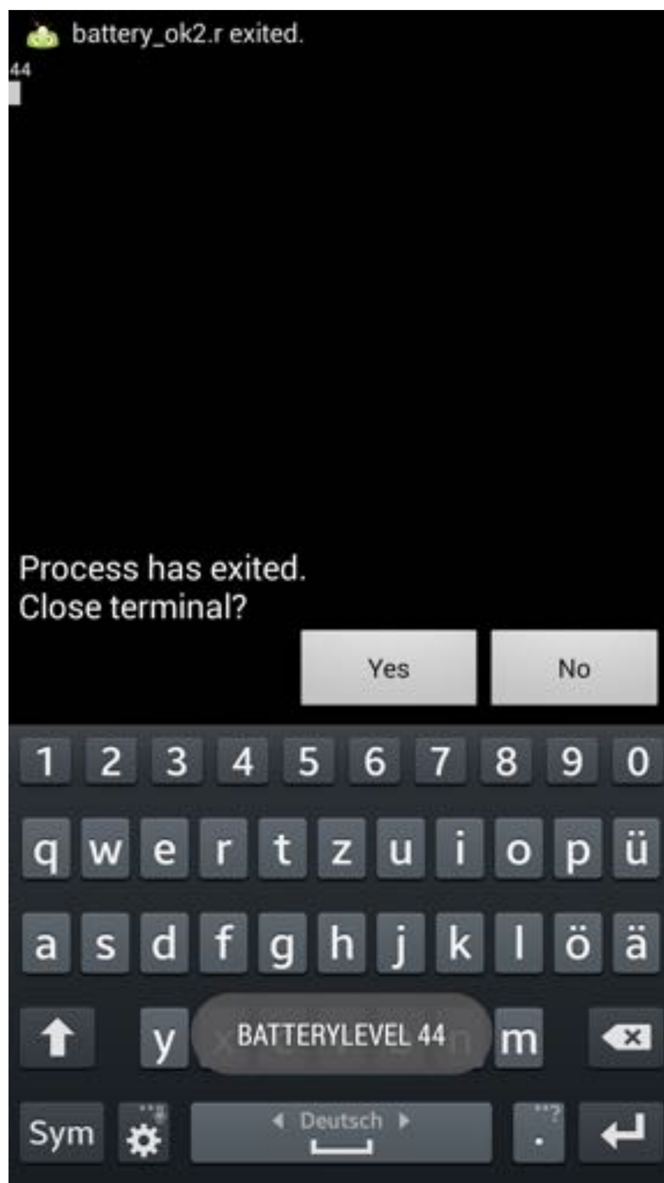


Figure 9: Battery status - Output

The output can be seen in Figure 9. On top of the picture is the previously mentioned command line output whereas the Toast stating the battery level can be found on the bottom of the picture.

2.5 Hi, Bluetooth!

Bluetooth functionality is the main part of this example script. Two devices are needed to run this script. A bluetooth server is started first in order to enable a bluetooth client to connect to the server. The server is then able to send messages to the client. The script code of the bluetooth server can be found in Listing 5 and the bluetooth client's code in Listing 6.

```
1  /* Server */
2  call import "android.r"
3  call AndroidInit
4  call toggleBluetoothState true
5  say "bluetooth is on!"
6  call bluetoothMakeDiscoverable 300
7  say "now discoverable"
8  call bluetoothAccept "457807c0-4897-11df-9879-0800200c9a66", 0
9  say "connected!"
10 message = dialogGetInput("Your Message","Please enter message:")
11 call bluetoothWrite message
12 call sleep 10
```

Listing 5: Bluetooth Server

```
1  /* Client */
2  call import "android.r"
3  call AndroidInit
4  call toggleBluetoothState true
5  a = bluetoothConnect("457807c0-4897-11df-9879-0800200c9a66")
6  say a
7  call sleep 5
8  a = bluetoothRead(4096)
9  say a
10 pull .
```

Listing 6: Bluetooth Client

The bluetooth server has to be started first before the client can be started as well. So, the first two lines of code are again necessary for initialisation purposes.

Right after that, the `toggleBluetoothState` function is called, passing true as argument.
Functionality: This function toggles the phone's bluetooth state on and off.
Parameters: This function has two optional input parameters.

- **Boolean enabled**: This argument specifies whether bluetooth should be enabled or disabled.
- **Boolean prompt**: This argument can be either true or false and defines whether the user has to be prompted to confirm to change the phone's bluetooth state. By default this value is true.

The message box prompting the user can be seen in Figure 10.

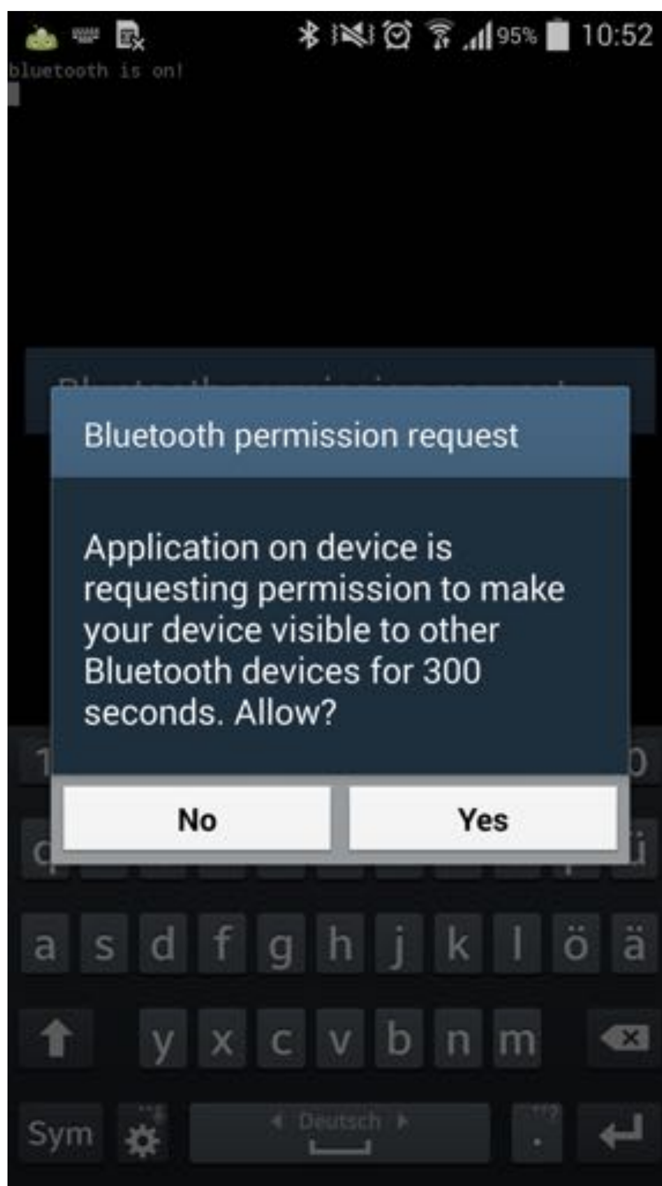


Figure 10: Bluetooth permission request

After the message "bluetooth is on!" is printed, the script calls the `bluetoothMakeDiscoverable` function passing the value 300 as an argument.
Functionality: This function makes the device discoverable for bluetooth connections.
Parameters: This function has only one optional parameter.

- **Integer duration**: This value specifies the duration in which the device should be discoverable by other devices.

Next, a message is printed to inform the user about the successful operation.

After that, the `bluetoothAccept` function is called.
Functionality: This function looks for a bluetooth connection, accepts it and blocks until the connection is either established or has failed.
Parameters: This function has two optional parameters.

- **String uuid**: This argument is an identification text. This string has to be identical to the string parameter of the `bluetoothConnect` function used by the client. By default it is set to *457807c0-4897-11df-9879-0800200c9a66*.
- **Integer timeout**: This value specifies how long the script is supposed to wait for a new connection. The default value is 0 and means the script is supposed to wait forever.

Again, an information message is printed for the user as in Figure 11.

Figure 11: Bluetooth server - Information messages

In the next step a message is retrieved by calling the `dialogGetInput` function as described in Section 2.1 This can be seen in Figure 12.

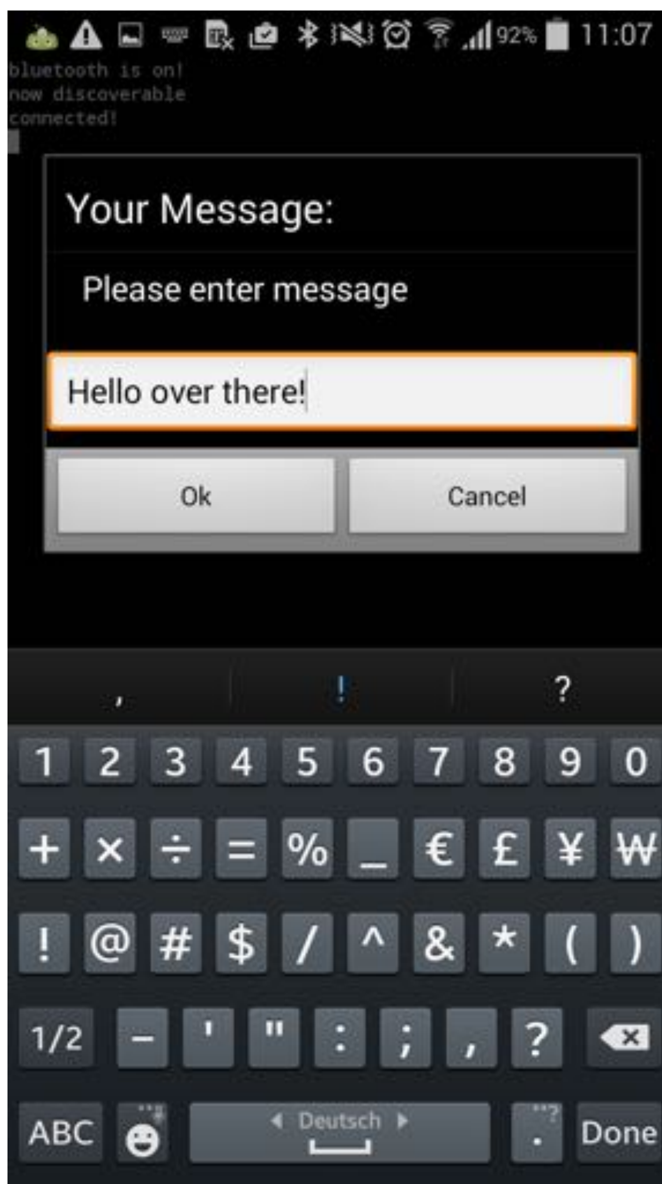


Figure 12: Bluetooth server - Message

Subsequently, the script calls the `bluetoothWrite` function passing over the message "Hello over there!" entered by the user.
Functionality: This function sends ASCII characters over the bluetooth connection that is open at the moment.
Parameters: This function has two parameters.

- **String ascii**: This parameter is the text to be sent.
- **String connID**: This parameter is the connection id. This string has to be identical to the string parameter of the `bluetoothAccept` function used by the server.

Rexx's `sleep` function is used to make the script wait for 10 seconds. This is necessary to keep the connection open long enough for the client to read the message sent by

the server.

The first two lines in Listing 6 are again needed for the initialization. Next, bluetooth is turned on using the `toggleBluetoothState` function as described above in this section. Then, the `bluetoothConnect` function is called to open a connection to the bluetooth server. An information message is printed before the script waits for 5 seconds, invoked by Rexx's `sleep` function, for the server to send a message.

This message is retrieved by the `bluetoothRead` function.
Functionality: This function reads characters, bufferSize at most.
Parameters: Both parameters are optional.

- **Integer bufferSize**: This argument specifies the maximum size of the message (ASCII characters) to be read.
- **String connID**: This argument is the connection ID.

At last, the message is printed as in Figure 13.

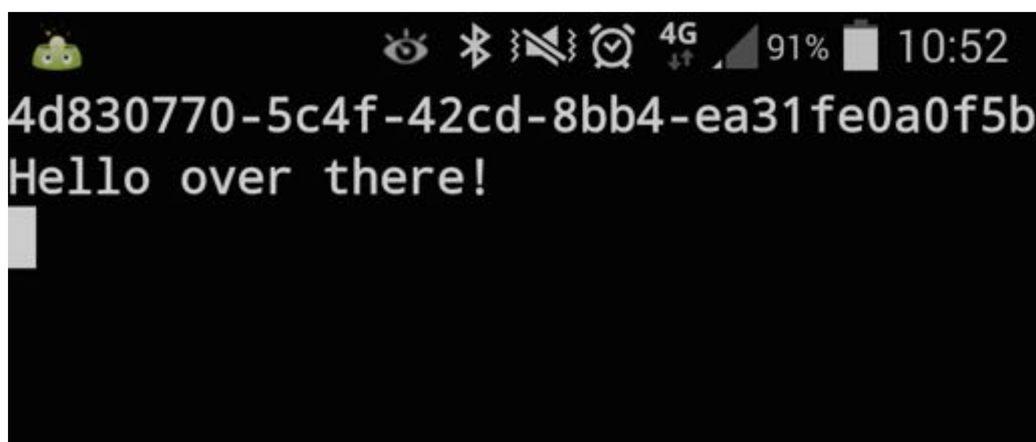


Figure 13: Bluetooth client - Received message

2.6 Look @ Maps

```
1  call import "android.r"
2  call AndroidInit
3
4  call mapZoom 1
5  call mapGoTo "usa"
6  call sleep 7
7  call mapGoTo "wu wien"
8  call sleep 7
9  call mapZoom 20
10 call sleep 7
11 call mapGoTo "tu wien"
12 call sleep 7
13 call mapZoom 10
14
15 exit
16
17 mapGoTo:
18   parse arg location
19   call startActivity "android.intent.action.VIEW", "geo:0,0?q="location
20   return
21
22 mapZoom:
23   parse arg zoom
24   call startActivity "android.intent.action.VIEW", "geo:0,0?z="zoom
25   return
```

Listing 7: Look @ Maps

First of all, initialisation is performed.

After that, the `mapZoom` function is called, which is a user defined Rexx function.
Functionality: An activity is started using the `startActivity` function.
Parameters: The function receives one parameter.

- **Integer zoom**: This is the level at which Maps will zoom. Possible values range from 0 to 20.

The `startActivity` function can start any Android Activity and has 7 arguments of which 6 are optional.

Functionality: Starts any Android Activity.
Parameters: The function receives 7 parameters.

- **String action**: The Android action to be performed.
- **String uri**: This is a uniform resource identifier which is necessary for certain operations (e.g. picking a contact).
- **String type**: This specifies the MIME type or subtype of the URI.
- **JSONObject extras**: Extras can be added to the Intent using this JSONObject.
- **Boolean wait**: This argument specifies whether the system has to wait until the user exits the started Activity.
- **String package name**: The name of the package.
- **String classname**: The name of the class.

After that, the `mapGoTo` function is called. This function is basically the same as the `mapZoom` function, the only difference is that instead of a zoom level a location is provided. This is necessary because it is not possible to pass both arguments at once. It would not cause an error but one of the arguments would simple be ignored. So after going to the USA, the script waits for 7 seconds and then moves its view to the WU in Vienna, which can be seen in Figure 14. Note that, at the time of writing (fall 2014) Google Maps still shows the former location of WU Vienna (Augasse 2) although it actually is located at Welthandelsplatz 1. After waiting for another 7 seconds, it is zoomed in to the maximum level. The location is changed to TU Vienna after waiting another 7 seconds. Then after waiting again, it is zoomed out (to level 10).

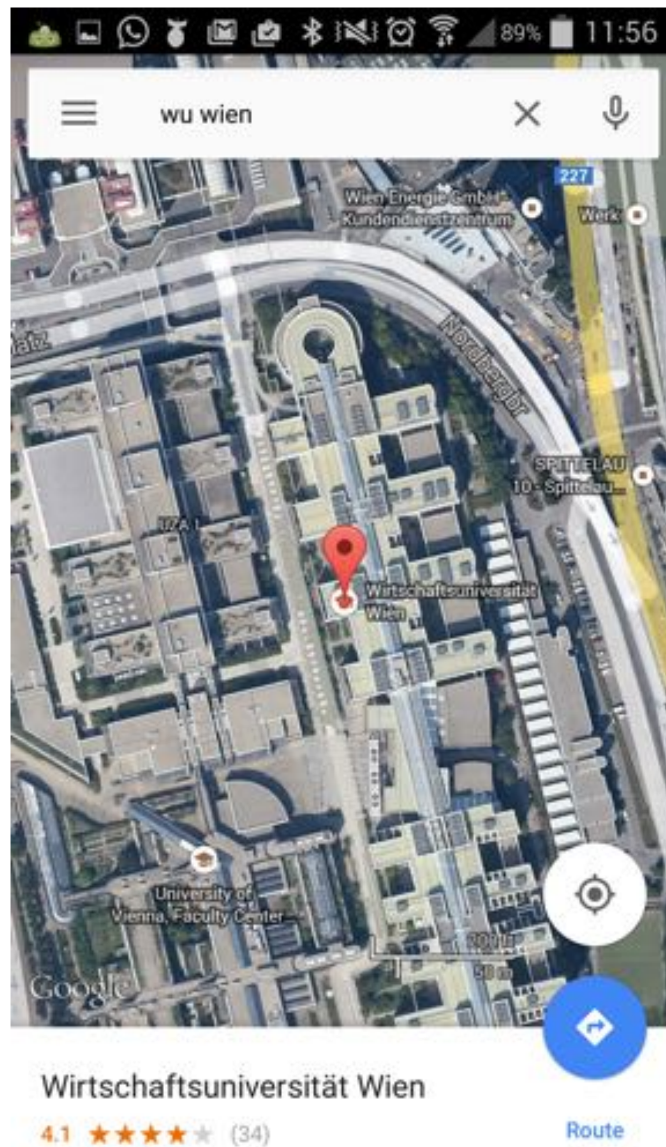


Figure 14: Google Maps showing WU Vienna

2.7 Phone-info

The next short script is supposed to show the use of some useful functions in order to obtain information about the phone, the network operator and also about the SIM card. The script code can be found in Listing 8 and the result can be found in Figure 15.

```
call import "android.r"
call AndroidInit

say "DeviceId:" getDeviceId()
say "CellLocation:" getCellLocation()
say "DeviceSoftwareVersion:" getDeviceSoftwareVersion()
say "NeighboringCellInfo:" getNeighboringCellInfo()
say "NetworkOperator:" getNetworkOperator()
say "NetworkOperatorName:" getNetworkOperatorName()
say "PhoneType:" getPhoneType()
say "SimCountryIso:" getSimCountryIso()
say "SimOperator:" getSimOperator()
say "SimOperatorName:" getSimOperatorName()
say "SimSerialNumber:" getSimSerialNumber()
say "SimState:" getSimState()
say "SubscriberId:" getSubscriberId()
```

Listing 8: Phone-info

The first two lines of the script perform the initialisation. After that, many different functions are called, always printing the essential part of the name of the function first, so that users have a clue about what the results really are.

The functions called in Listing 8 are now briefly described. All of them do not have any input parameters, therefore parameters are not mentioned for each function separately like they are in the previous examples.

- **getDeviceId():** Returns the device's unique id, e.g. IMEI and MEID numbers.
- **getCellLocation():** Returns the location of the current cell.
- **getDeviceSoftwareVersion():** Returns the number of the software version running on the device, e.g. IMEI/SV.
- **getNeighboringCellInfo():** Returns information about the device's neighboring cell.
- **getNetworkOperator():** Returns the numeric name of the current operator.
- **getNetworkOperatorName():** Returns the alphabetic name of the current operator.
- **getPhoneTyp():** Returns the type of the device, e.g. GSM.

- **getSimCountryIso():** Returns the ISO country code of the SIM provider.
- **getSimoperator():** Returns the numerical code of the SIM provider using 5 or 6 decimal digits.
- **getSimOperatorName():** Returns Service Provider Name (SPN).
- **getSimSerialNumber():** Returns the serial number of the SIM, in case it is available.
- **getSimState():** Returns the state of the SIM cards.
- **getSubscriberId():** Returns the unique subscriber ID. e.g. IMSI.

The output of the function calls can be seen in Figure 15, `getSimOperatorName` returns an empty string.

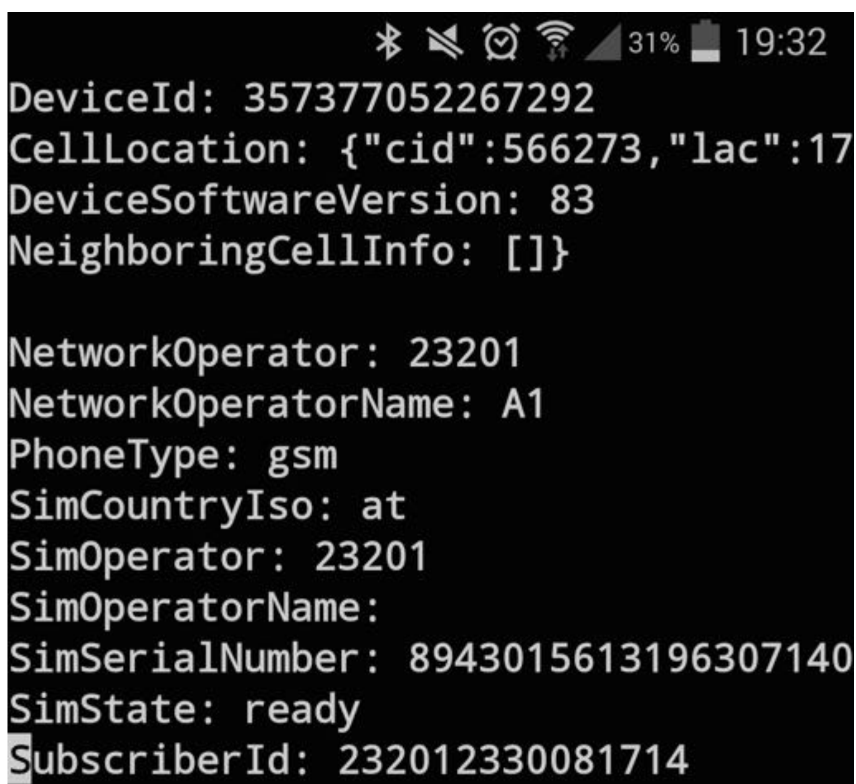


Figure 15: Phone-info

2.8 Scan My Barcode

This short script scans a barcode of a book and opens the corresponding Google Books page. It is required that a barcode scan application is installed before the script is run. Listing 9 shows the script code.

```
call import "android.r"
call AndroidInit

code = scanBarcode()
isbn = jsonDecode(code,"SCAN_RESULT")
url ="http://books.google.com?q="isbn
call startActivity "android.intent.action.VIEW", url

exit

jsonDecode:
  parse arg json, key
  key = key||'":"'
  parse var json . (key) value '"' .
  return value
```

Listing 9: Scan my Barcode

After initialisation, the `scanBarcode` function is called and the result saved.
Functionality: The function starts a barcode scanner.
Parameters: The function does not receive any arguments. It returns the result Intent in Map representation (which is a JSON object).

In a next step the `jsonDecode` function, a user defined Rexx function, is called and the result of the code scan as well as the key of the desired value is passed. The JSON string is parsed to extract the desired value. The extracted value is then returned. An url is built by adding the extracted ISBN number to an already existing link. This url is than opened via the `startActivity` function explained in Section 2.6.

3 BRexx versus Rexxoid

Rexxoid and BRexx are both interpreters of Rexx for Android. While BRexx builds on the scripting layer for Android (SL4A), Rexxoid does not make use of it. As only BRexx uses this functionality, SL4A is considered to be a part of BRexx for the comparison. In this section they will be compared and major differences highlighted.

Both applications were tested on the same device, Samsung Galaxy S4 (model number GT-I9505) running Android version 4.4.2. Therefore there should be no differences owing to the used device. All statements refer to the time of writing, which is fall 2014.

This section was written in cooperation with Julian Reindorf [see Rein14].

3.1 Comparison

Installation The Rexxoid application can be downloaded directly from the Google Play store. Android devices install the downloaded application automatically and after that the installation process is finished. BRexx needs two applications, SL4A and BRexx itself, neither of them can be downloaded from the Google Play store. They have to be obtained from external sources. This requires also the permission to install applications from third parties, which is not necessary for Rexxoid.

Example repository Rexxoid comes with a few preinstalled examples. Of these few examples only one uses Android functionality. Also there are no further examples for Rexxoid on the web which use Android functionality. In comparison, BRexx also comes with only a few preinstalled examples. However, some more can be found on the web. Admittedly, those are example scripts in other languages (e.g. in Python or JavaScript), but as they also use SL4A functions they can be adapted quite easily.

Functionality The functionality of Rexxoid is very limited. Besides "normal" Rexx code, only Android shell commands can be used. Hardly any predefined functions are available. BRexx, on the other hand, offers a great variety of functions. Almost any Android component can be addressed in BRexx (e.g. sensors can be utilised). Additionally, in contrast to Rexxoid, BRexx scripts can run in the background, which leads to even further fields of application.

Although both applications come with a small set of granted Android permissions, for some functions additional permission are necessary. Adding permissions is rather

effortful in both cases, it can only be done by editing the source code of the application (specifically, the manifest file) and therefore requires a reinstallation. BRexx has the advantage that written scripts are not deleted in this process, while Rexxoid's need to be saved in advance.

Usability Rexxoid's user interface is very slim and the navigation is rather inconvenient (e.g. the script has to be opened in editing mode before it can be executed). BRexx's interface is more sophisticated and offers handy options (e.g. Save & Run). Also the menus are very well structured and useful. This makes getting started very easy.

Performance Both applications come with the same script to measure their performance (Average REXX clauses-per-seconds by Mike Cowlishaw). When executing the script, Rexxoid ends up with about 450.000 clauses per second, whereas BRexx reaches about 1.1 million. It can be seen that BRexx is about twice as fast as Rexxoid.

Community Rexxoid's community consists only of a handful of people. BRexx's community is actually also very little, but because of the big SL4A community support for many of the upcoming problems can be found in this community as well.

Documentation There is no documentation of Rexxoid. BRexx does not offer any documentation as well. But at least SL4A offers a list of available functions and their arguments. Unfortunately, there is no information on dependencies of the functions (i.e. function x has to be called before function y or y will not work). Nevertheless, the built in API browser is comfortable to use.

Readability Reading and understanding Rexxoid scripts is quite difficult. Android shell commands are very long and not self-explanatory. Due to very meaningful function naming in SL4A, BRexx scripts are easy to read and understand.

Debugging Debugging in Rexxoid is easier than in BRexx. Exceptions thrown by the operating system or by Rexxoid itself are displayed in Rexxoid's default output console. Syntactical errors are noticed immediately and prevent execution of the script. BRexx does not display any errors by default. After execution of a script, the unfiltered logcat output can be displayed (which is really long and not overseeable). So the best way

of displaying errors and debugging is to connect the device to a computer and use the Android Debug Bridge to display a filtered logcat output.

Software updates The newest version of Rexxoid is from September 2014, while BRexx's newest version is from March 2013. Rexxoid is being developed continiously, BRexx has longer release cycles.

3.2 Recommendation

Table 1 summarises the results of Section 3.1. The application which performs better in a specific aspect is marked with a plus sign (+), the less well performing application marked with a minus sign (–). If they are equal, a tilde is used (~).

Aspect	Rexxoid	BRexx
Installation	+	−
Example repository	−	+
Functionality	−	+
Usability	−	+
Performance	−	+
Community	−	+
Documentation	−	+
Readability	−	+
Debugging	+	−
Software updates	~	~

Table 1: Comparison of Rexxoid and BRexx

As Table 1 indicates, BRexx is the better choice in almost any perspective. Maybe Rexxoid improves and enriches its feature set in the future, but currently the clear recommendation is BRexx.

4 Conclusion

SL4A is a very elaborated application enabling users to execute scripting languages on Android devices, which otherwise would not be possible without rooting the device. BRexx allows the implementation of scripts using the simple and yet powerful language Rexx.

Although the beginning is hard and time consuming, writing scripts becomes easier (but still not easy) over the time. In case problems arise, example scripts in other languages can be of use. It would be easier and more efficient if there were (more) SL4A examples in BRexx, but examples written in other languages can be a good starting point.

Especially if scripts are supposed to handle minor tasks, BRexx is of great advantage compared to native Android applications. A few lines of code are sufficient to send SMS or make use of the device's sensors. This allows developers who are not familiar with Android programming to still create useful scripts using Android functionality.

As discussed, compared to Rexxoid, another approach of running Rexx on Android, BRexx is more advanced, easier to learn and more user-friendly as well.

Hopefully this work encourages others to try out BRexx as well and help to enlarge the example set. Nutshell examples help developers to save a lot of time and effort, therefore the whole community can benefit from new scripts.

References

[Brex14] BRexx.: Downloads. http://pceet075.cern.ch/bnv/brexx, Accessed on 2014-12-23.

[Flat13] Flatscher, Rony G.: Introduction to REXX and ooRexx. Facultas, 2013.

[Goog14] Google.: Android scripting. https://code.google.com/p/android-scripting/, Accessed on 2014-12-23.

[Inte14] International Data Corporation.: Smartphone OS Market Share, Q3 2014. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, Accessed on 2014-12-01.

[Meie14] Meier, Reto.: Android App-Entwicklung: Die Gebrauchsanleitung für Programmierer. John Wiley & Sons, 2014.

[Open14] Open Handset Alliance.: Overview. http://www.openhandsetalliance.com/oha_overview.html, Accessed on 2014-12-01.

[Rein14] Reindorf, Julian.: Rexxoid: Running Rexx on Android Systems. Vienna University of Economics and Business, Seminar Thesis, 2014.

[Stau13] Staudemeyer, Jörg.: Android-Programmierung kurz & gut. O'Reilly Germany, 2013.